

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

响应式架构

消息模式Actor实现与Scala、Akka应用集成

[美] **Vaughn Vernon** 著

苏宝龙 译

Akka项目创始人Jonas Bonér为本书作序

Reactive Messaging Patterns *with the* Actor Model

Applications and Integration in Scala and Akka



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

响应式架构

消息模式Actor实现与Scala、Akka应用集成

[美] Vaughn Vernon 著

苏宝龙 译

Reactive Messaging Patterns

with the

Actor Model

Applications and Integration in Scala and Akka

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

通过Actor模型使用响应式消息传输模式,可编写出具有高性能、高响应性、高可伸缩性和高韧性的并发应用程序。本书由10章构成,详细介绍了使用Actor模型中的响应式消息传输模式的理论和实用技巧。其中包括:Actor模型和响应式软件的主要概念、Scala语言的基础知识、Akka框架与Akka集群功能、Actor模型中的通道机制和技术、降低消息源与消息目的地之间耦合性的方式、持久化Actor对象和幂等接收者。附录A中还介绍了通过.NET平台和C#语言使用Actor模型的方式。

在企业中任职的软件架构师和开发者,以及任何对Actor模型感兴趣并渴望提高自身技术和价值的软件开发者,均适合阅读本书。

Authorized translation from the English language edition, entitled Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka, by Vaughn Vernon, published by Pearson Education, Inc., publishing as Addison-Wesley Professional, Copyright © 2016 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright © 2016.

本书简体中文版专有出版权由Pearson Education培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有Pearson Education 培生教育出版集团激光防伪标签,无标签者不得销售。

版权贸易合同登记号 图字:01-2015-7885

图书在版编目(CIP)数据

响应式架构:消息模式Actor实现与Scala、Akka应用集成/(美)沃恩·弗农(Vaughn Vernon)著;苏宝龙译.—北京:电子工业出版社,2016.7

书名原文:Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka
ISBN 978-7-121-29113-5

I. ①响… II. ①沃… ②苏… III. ①程序语言-程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2016)第137490号

策划编辑:张春雨

责任编辑:刘 舫

印 刷:北京中新伟业印刷有限公司

装 订:北京中新伟业印刷有限公司

出版发行:电子工业出版社

北京市海淀区万寿路173信箱

邮编:100036

开 本:787×980 1/16

印张:27.5 字数:555千字

版 次:2016年7月第1版

印 次:2016年7月第1次印刷

定 价:99.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888,88258888

质量投诉请发邮件至zltts@phei.com.cn,盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819 faq@phei.com.cn。

推荐序

献给我最亲爱的 Nicole 和 Tristan,

你们的爱与支持是我前进的动力!

推荐序

终于有一本围绕企业应用和架构来讲解 Actor 模型和 Akka 的书了。很期待这类书的出现，希望能引领 Actor 模型开始向企业应用的“回归”。

本书作者 Vaughn Vernon 恰好也是《实现领域驱动设计》(*Implementing Domain Driven Design*)一书的作者，这在某种程度上印证了，近十来年的“领域驱动设计(DDD)”理想在 Actor 模型和 Akka 上终于找到了现实的技术实现。

DDD 希望能在业务领域层面就把模型和逻辑设计清楚(业务模型和逻辑是最稳定的)，并一一对应到实现中。或者说，领域有什么，实现中才应该有，也应该有。但由于计算机性能限制、语言实现难度等方面的原因，这一理想在现实中一直没能得到很大程度的实现。

而近二十年来，Java 及其生态一直占据着企业应用领域的主导地位。尤其从 20 世纪 90 年代末以来，J2EE 规范(现在叫 Java EE)也试图围绕业务领域，为企业应用提供从建模到分层，涵盖事务、持久化、分布式的整套解决方案，并提出 Entity Bean、Session Bean 及 Message Bean 等试图对应到业务领域的模型。

遗憾的是，基于当时的技术，Java EE 也并没有很好地实现初衷，这在我看来至少有以下几方面的原因：

- EJB 规范起初的一个主要价值——对分布式应用进行事务管理，在实践中几乎很少被使用，反倒引入了整个架构的复杂性。
- Entity Bean 不能在集群中分片部署，而这本应是分布式系统最需要解决的问题之一。
- 只有 Message Bean 是异步的，但它却不是 Entity Bean。这意味着系统很难在时间维度解耦。

2005 年前后，Hibernate、Spring 等技术逐渐兴起，以轻量化的角度切入了企业应用领域，并在互联网领域异军突起。

对于 Spring + Hibernate 的方案，保存在持久层的业务实体的数据/状态

需要反复被业务逻辑存取。为了解决这个性能瓶颈，不得不引入 Ehcache、Memcached、Redis 等中间缓存。这样，在数据扩张时，为了解决这些缓存数据的分片（sharding）问题，这些缓存方案还需要进一步引入和实现集群分片的支持，这带来了复杂性。可即便如此，它解决的是缓存数据的分布，而并没有解决 Beans 本身的分布。Beans 仍然受限于 Bean 容器的缓存大小，而不得不经常去中间缓存甚至持久层要数据。

那么，在企业应用领域，Actor 模型能带来更合适的解决范式吗？

Carl Hewitt 在 1973 年对 Actor 模型进行了如下定义：“Actor 模型是一个把‘Actor’作为‘并发数字计算的通用原语’的数学理论。”。这个定义跟我常说的“Actor 是最适合并行计算的最小颗粒”是相通的。

Actor 是异步驱动、可以并行和分布式部署及运行的最小颗粒。也即，它可以被分配、分布、调度到不同的 CPU、不同的节点，乃至不同的时间片上运行，而不影响最终的结果。因此，Actor 是在空间（分布式）和时间（异步驱动）上解耦的。

Akka 是 Lightbend（前身是 Typesafe）公司在 JVM 上的 Actor 模型实现，它同时是一个可扩展、引入了多种分布式范式的框架。而且，Akka 2.3.0 开始支持带状态的 Actors 的分片集群，以及根据 journal/snapshot 形式对事件流和状态快照实施持久化和回放。

Akka 的 Actor 模式本身可以保证在单个 Actor 实例中每个行为的原子性，并行的粒度可以细化到单个 Actor 实例。也即，当行为被封装在一个 Actor 实例中时，该行为不会阻塞其他 Actors 实例的行为，从而很难出现整个系统被阻塞的情况。

从 EJB 角度看，Akka Actor 提供了什么对应的角色呢？

首先，从各类 Bean 的角色看。Akka Actor 支持持久化，所以有一类 Actor 可以设成“Entity Bean”；Actor 实例可以维护自己的状态，所以它也可以是“Stateful Session Bean”；而不需要关心其状态的 Actor，自然可以担当“Stateless Session Bean”；最后，对 Actor 的存取、调用，都是通过异步的消息传递来实现的，因此，它们都是“Message Bean”。

其次，从架构层面看。Actor 能同时担当实体 Beans 和中间缓存的角色，并且是异步驱动的，且具备分片集群下的水平扩展能力。而 akka-persistent 进一步将持久化、HA 一并细粒度地实现了。与 SSH/Java EE 相比，Actor 减少了数据反复在各种形态（数据库、缓存、业务层中的实体对象实例）间转化的消耗，减少了线程阻塞的消耗，并提供了一致的并行和分布式机制。

再次，从业务领域看。Actor 可以非常自然地直接对应到业务实体。某类 Actor 的一个实例可以是一个人、一个物品、一部设备，等等。这些实体 Actor 都

是通过接收命令或者事件，来驱动完成一次状态的变化或者完成一次任务会话。Akka 的每个 Actor 仍由自己的 scheduler 来完成各类定时任务，也可以理解为它们同时可以由时钟事件来驱动。而一次任务会话也可以抽象为一个会话 Actor 的实例，它跟踪会话的进度、事务状态、发送事件等。

最后，就像前面提到的，对 Actor 实例的存取、调用是通过异步的消息传递来实现的，这带来一种担心：性能代价会不会很高？

这确实是很多年前，采用异步消息驱动来设计、实现系统的架构师、程序员的最大障碍，因为那时的机器不仅性能有限，多核的物理机制也少见且昂贵，而且多线程的切换代价高昂。

但对于现在的计算机而言，多核处处可见，CPU 的计算能力、线程的调度与切换能力也有了极大的提高，上面这些问题已经不再是障碍。比如，Akka 在单核的 CPU 上，每秒可以处理的异步消息数是 5000 万以上。尤其是，现代 JVM 的线程切换时间已经在微秒级，线程切换的代价变得非常小。小到在大量场景下，采用异步处理所带来的整体性能和效率的提升已经足够将其代价忽略。

还有一种担心来自：用 Akka 会引入复杂的架构吗？

从 DDD 的理念和实践来看，恰恰相反，因为处处都是同样的模型（Actor、Async Message Driven、Event Streaming），系统实际上更具一致性和弹性（包括对需求的弹性）。

这些年，我本人一直在尝试将 Actor 模式及 Akka 用于企业应用，包括证券交易及计算领域，以及在豌豆荚实现的若干实时系统。实际上最近十年做来做去无非是同一件事：将现实（直接）映射到计算机体系的 Actor 模型。而实践效果充分验证了，采用异步消息驱动以及小粒度的并行和持久化机制，在性能和内存的使用上都不是问题。更重要的是，模型及架构与领域的自然对应大大降低了系统进化和维护的成本。而得益于 Akka 的集群、高可用及事件溯源（Event-Sourcing）的持久化机制，这几个系统也几乎都能无故障地持续运行。运行时间最长的一个，超过了两年没有重启。

本书的内容是基于 Akka 2.3.4 版本的，这个版本包含了 Akka 框架主要的功能和实现（包括 sharding 和 persistent），非常新且全面。而且作为一个长期从事企业应用领域的设计和实现的专家，作者非常熟悉在企业业务领域需要用到的知识和术语以及思维方式，并很好地融入了 Akka 的实践。

译者序

1965 年 Intel 的创始人戈登·摩尔发现了摩尔定律，50 多年来，计算机的性能一直遵循摩尔定律迅猛发展：CPU 可容纳的晶体管数目，每隔约 18 个月便会增加一倍，性能也将提升一倍。如今 CPU 中晶体管的数量以指数形式增长的迅猛势头似乎要走到尽头了。而计算机性能的另一要素——CPU 主频速度的提高，早在 2003 年就开始急剧下降。计算机的性能无法迅猛增长的同时，人们的需求却仍旧以指数形式增长，供求矛盾日益尖锐。

传统提高 CPU 性能的技术已经被多核和超线程技术取代。事实证明，硬件工程师再也不能独自承担提高计算机性能的重任了。当前硬件工程师确实能够设计出含有 288 个核心的 CPU，但如果该 CPU 没有被用于运行相应的并发程序，这个含有 288 个核心的 CPU 只能被当作单核 CPU 使用。

该是软件工程师挺身而出勇挑重担的时候了。但是，使用传统的并发技术（如线程、锁和监控器等）开发软件会遇到许多难以克服的难题。例如，到软件开发过程的末尾阶段，客户提出增加功能的要求，或者需要对某个（些）功能进行改进，就不得不重新调整线程的分配和几乎所有并发分支。这些工作量可能不比重新开发一个新的软件少多少，甚至可能会比开发新的软件更加困难。因此，传统并发技术注定不能担任当前软件开发工作的主角。开发者们迫切需要的是高级并发编程技术。

Carl Hewitt 博士早在 20 世纪 70 年代初就发明了 Actor 模型，这种优秀的高级并发编程思想超越了 Carl Hewitt 博士所处的时代。但当时功能最强大的处理器也无法将该理论付诸实践。直到多核处理器、云计算、移动设备和互联网无处不在的今天，Actor 模型才重新焕发了青春。

Actor 模型拥有下列优点：

1. 大幅度降低应用程序内部的耦合性。
2. Actor 模型的消息传递形式简化了并行程序的开发工作，使开发人员无须

与并发编程基础元素打交道。

3. 在高动态环境中, Actor 模型既可以利用顺序编程技巧, 也可以利用函数编程技巧。
4. Actor 模型可以解决许多并发编程难题, 如死锁、活锁、互斥体等。
5. Actor 模型能够大幅度提高调用方法的安全性和速度。

凭借 Actor 模型的这些优势, 通过响应式消息传输模式, 开发者能够开发出具有高性能、高响应性、高可伸缩性和高韧性的并发应用程序。

本书的作者 Vaughn Vernon 是一位资深的软件开发, 并且是一位简化软件设计和实现思想的领袖人物。他在本书中使用了大量的实践案例, 这些范例程序既有实用性也有启迪性, 深入浅出地讲解了使用 Actor 模型通过 Scala 语言和 Akka 框架, 编写响应式应用程序的理论和实用技巧。

翻译前沿计算机科学书籍的工作并不轻松, 也不是单独一个人能够完成的。在此我要感谢电子工业出版社计算机出版分社的张春雨等编辑对本书提供的帮助。此外, 石浩、孙顾、徐颖、朱晶晶、沈骏杰、何志颖、许诗怡、马佳妮、尹晓婷、徐雯、郭昕、陆迎明和孙艳婷等也参与了本书的翻译工作。

因时间仓促, 译者水平有限, 本书的错漏之处欢迎广大读者朋友们批评指正。

序

20 世纪 70 年代初, Carl Hewitt 发明了 Actor 模型, 他超越了自己所处的时代。他通过 Actor 概念, 定义了一个含有不确定性的计算模型(假设所有计算操作都是通过异步方式执行的)。该模型使用并发处理模式和有稳定的状态独立处理过程的概念, 全方位地降低了 Actor 对象的耦合性, 并使之支持分布式和移动架构。

当前, 软件行业已经跟上了 Carl Hewitt 的创新思路; 多核处理器、云计算、移动设备和互联网都成为常见的事物。这从根本上改变了软件行业, 而且使创建并发模型和分布式处理基础理论的需求变得更为迫切。我相信 Actor 模型能够成为我们迫切需要的坚实理论基础, 使我们能够通过具有响应性、韧性和弹性的响应式编程原则, 创建复杂的分布式系统以应对当前的挑战。这就是我编写 Akka 框架的原因: 将 Actor 模型的强大功能交到普通开发者手中。

看到 Vaughn Vernon 撰写的这本书我感到非常兴奋。这本书介绍了大家都需要了解的知识——将 Actor 对象与传统的企业级消息传输系统连接起来, 以及使用 Actor 对象创建响应式应用程序的方式。我喜欢这本书仅依赖 Akka 框架基础功能(是 Actor 模型而不是 Akka 框架中的高级库)来介绍高级消息传输和通信模式的方式。即使 Actor 模型仅是一种低等级的计算模型, 但看到使用它可以通过简单直观的方式实现功能强大且多样的消息传输模式, 确实是一件令人赏心悦目的事情。一旦你了解了基础的编程思路, 就能够向其中添加高级工具和技巧。

这本书还介绍了许多形式化和命名模式, 这是 Akka 社区成员通过数年的研究探索和反复改进获得的成果。这使我回忆起几年前, 读到 Gregor Hohpe 和 Bobby Woolf 撰写的经典著作 *Enterprise Integration Patterns*[EIP] 时的惊喜之情。我对 Vaughn Vernon 能够继承这本经典著作的精髓, 并对其做了全新的诠释感到很高兴。但我认为这本书最重大的贡献在于, 它并没有止步于前人探索过的区域, 而是为 Actor 对象的消息传输操作定义了一种独特的模式化语言。这使我们能够

使用专业术语来思考、讨论和交流 Actor 对象传输消息的模式和编程思路。

不论你是初学者还是资深的编程高手，这本书都能够为你提供重要帮助。我希望你能够和我一样与它成为好朋友。

Jonas Bonér

Akka 项目的创始人

前言

当前，许多软件项目折戟沉沙。各种调查和研究报告表明，软件项目的失败比例为 30% 至 50%。该统计数字还不包括那些已经被交付使用，但没有达到某些成功标准的软件项目。当然，该统计数字中包含了企业级的软件项目。上述数据摘自 *Dr. Dobbs's Journal*[DDJ] 杂志中由 Scott Ambler[Amblysoft] 撰写的调研报告 *Chaos Report*[Chaos Report]。

与此同时，许多公司使用 Scala 语言和 Akka 框架在突破性能限制和获得可伸缩性方面，取得了令人瞩目的成功 [WhitePages]。这些成功不仅代表软件项目的成功，还代表了在非功能性需求方面取得的成功。当然，这些成功不是仅通过 Scala 语言和 Akka 框架获取的，但同时也不应抹杀 Scala 语言和 Akka 框架在这些成功中所起的重要作用。我确信开发者们是根据他们选择的平台使用这些工具的，这也是他们获得成功的关键。

几年来，我一直希望将 Scala 语言和 Akka 框架引荐给数量众多的想要获取上述成功的企业。本书的目标是使你熟悉 Actor 模型并通过 Scala 语言和 Akka 框架实现 Actor 模型的方式。此外，我相信许多企业级软件架构师和开发者已经通过 Gregor Hohpe 和 Bobby Woolf 撰写的经典著作 *Enterprise Integration Patterns*[EIP] 获得了启迪。*Enterprise Integration Patterns* 中介绍了 65 种整合模式，这些模式能够帮助开发团队将企业中的各种独立系统整合到一起。我认为通过 Actor 模型来使用这些整合模式，除了能够使这些模式非常得心应手外，还会使架构师和开发者获得如虎添翼的感觉。

当通过 Actor 模型使用这些整合模式时，与其他方式的主要差异是使用这些整合模式的原始动机。在通过 Actor 模型使用这些整合模式时，这些整合模式被用于开发新的应用程序，而不仅仅是被用于整合旧的应用程序。这是因为这些模式首先是消息传输模式，而后才是整合模式，而 Actor 模型的本质就是消息传输。当使用领域驱动设计 [DDD, IDDD] 方式时，你还会发现一些比较高级的整合模式

(如处理过程管理器)，可以帮助你通过直观的方式为突出的业务概念创建模型。

本书面向的读者

本书面向在企业中任职的软件架构师和开发者，以及任何对 Actor 模型感兴趣并渴望提高自身技术和价值的软件开发者。尽管本书着重介绍的是 Scala 语言和 Akka 框架，但在附录 A 中介绍了通过 .NET 平台和 C# 语言使用 Actor 模型的方式。

本书的主要内容

第 1 章介绍 Actor 模型和响应式软件的主要概念。第 2 章介绍 Scala 语言的基础知识，以及 Akka 框架与 Akka 集群功能。第 3 章介绍计算机性能的发展史和未来趋势，以及凭借 Scala 语言和 Akka 框架的可伸缩性，在企业级软件中通过 Actor 模型获得高性能和高可伸缩性的来龙去脉。

之后，本书共使用 7 章内容介绍 Actor 模型中的消息传输模式。第 4 章介绍基础消息传输模式，并概要介绍了后面 5 章的主要内容。第 5 章介绍基础的通道机制和几种通道技术，在使用这些通道技术应对各种应用程序开发和整合挑战时，每种通道技术都能够发挥本身独特的优势。第 6 章介绍消息的结构，指明消息中必须包含消息发送者与消息接收者进行通信的原因。第 7 章介绍降低消息源与消息目的地之间耦合性的方式，以及在消息路由器中放置合适的业务逻辑的方式。第 8 章详细介绍在应用程序开发和整合环境中，需要使用的各种消息转换方式。第 9 章介绍各种各样的消息端点，其中包括持久化 Actor 对象和幂等接收者。最后，第 10 章介绍更高级的编程模式、基础结构和调试工具。

本书的约定

本书主要介绍各种消息传输模式。你不必尝试一鼓作气地阅读所有这些模式，但通常你应该了解第 4 章至第 10 章的内容，以便知道各种模式的详细介绍在书中的具体位置。这样，当你需要掌握某种消息传输模式时，就能够迅速找到介绍它的内容。本书为每种消息传输模式都提供了一个标志性的配图，而且每种模式通常都会拥有插图和源代码（至少会有一个插图和一段代码），以详细介绍使用 Scala 语言和 Akka 框架实现该模式的方式。

本书介绍的各种消息传输模式，实际上一同构成了一种模式语言，这种模式语言由各种相互关联的开发模式和公式（如将基于内容的路由器与死信通道路由器配合使用）构成。在设计基于消息的应用程序和系统时，就可以使用这种模式语言。因此，了解各种模式之间的依赖关系十分必要，正是这些模式之间相互支持的关系构成了这门完整的语言。因此，本书提到某种模式时，会使用模式名称表示。

本书的第二个约定是使用插图表示消息传输模式和 Actor 模型的方式。我与 Typesafe 公司的 Roland Kuhn 和 Jamie Allen 一起制定了这些约定。这两个伙伴和我一起撰写了与本书题材类似的新书 *Reactive Design Patterns*，这本书很快会与大家见面。我想使用相同风格的插图介绍这两本书中的 Actor 模型，因此我找 Roland Kuhn 和 Jamie Allen 一起进行了讨论。下面是我们的讨论结果。

如图 P.1 所示，Actor 对象由圆圈代表，而且通常会将该对象的名称放在圆圈中。使用这种表示的多种原因之一是，Gul Agha 很久以前在他撰写的 *Actors: A Model of Concurrent Computation in Distributed Systems*[Agha, Gul] 一书中就使用了这种表示方式。

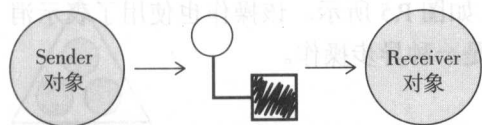


图 P.1 Actor 对象 Sender 向 Actor 对象 Receiver 发送了一条消息。

此外，*Enterprise Integration Patterns*[EIP] 一书也使用这种表示方式将消息表示为组件，因此我们延续了这种用法。带箭头的线条指明了传输消息的源头和目的地。通过图 P.2 和图 P.3 展示的方式，你可以将持久化 Actor 对象（长期存在的）和临时 Actor 对象（短期存在的）区分开。代表持久化 Actor 对象的圆圈是由实线绘制的。持久化是指该 Actor 对象会长期存在。持久化并不单指将 Actor 对象存储在硬盘中，也涵盖将 Actor 对象存储在其他永久性存储设备中。另一方面，代表临时 Actor 对象的圆圈是由虚线绘制的。临时是指该 Actor 对象仅会短期存在，在执行完特定任务后会被停止。



图 P.2 持久化 Actor 对象（长期存在的）。



图 P.3 临时 Actor 对象（短期存在的）。

如图 P.4 所示，一个 Actor 对象可以创建另一个 Actor 对象，这就形成了一种父子关系。一个小圆圈外套一个大圆圈和带箭头的线条代表了这种创建操作。其中带箭头的线条与表示消息传输的方式相同，这是因为创建子 Actor 对象的处理过程是一种异步操作。

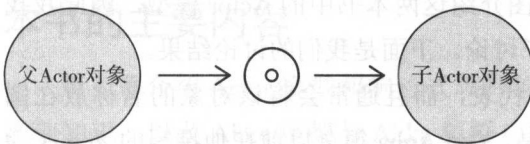


图 P.4 父 Actor 对象创建了一个子 Actor 对象。

Actor 对象的自行终止操作是通过向自己发送特殊的消息实现的。该特殊消息由外套圆圈的 X 和带箭头的线条代表，如图 P.5 所示。该操作也使用了表示消息传输的方式，因为这种自行终止操作也是一种异步操作。

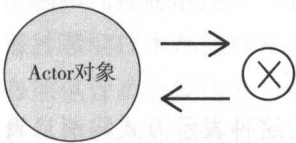


图 P.5 Actor 对象的自行终止操作。

一个 Actor 对象停止另一个 Actor 对象的操作也是通过相同的特殊消息实现的，但该消息是由一个 Actor 对象发送给另一 Actor 对象的。图 P.6 展示了父 Actor 对象停止它的子 Actor 对象的方式。

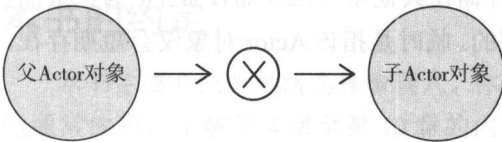


图 P.6 一个 Actor 对象停止另一个 Actor 对象。

通过统一建模语言（UML）顺序图可以表现 Actor 对象的生命周期，如图 P.7 所示。Actor 对象在其生命周期中收到的消息由小圆圈代表（与大头针类似）。你

必须认识到每条消息都是以异步方式接收的。

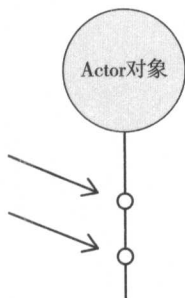


图 P.7 该 Actor 对象收到的两条消息展示了该 Actor 对象的生命轨迹。

父子层次结构由代表父 Actor 对象的圆圈和代表子 Actor 对象集合的三角形构成，如图 P.8 所示。



图 P.8 Actor 对象的父子层次结构。

一个 Actor 对象可以通过赋值或介绍操作认识另一个 Actor 对象。赋值操作是指在创建一个 Actor 对象时，将该 Actor 对象的引用赋予另一个 Actor 对象。另一方面，通过发送消息可以将一个 Actor 对象介绍给另一个 Actor 对象。

如图 P.9 所示，介绍操作由带箭头的虚线代表，该虚线会由被介绍的 Actor 对象指向被发送给另一个 Actor 对象的消息。在本例中，由父 Actor 对象创建的子 Actor 对象被介绍给了消息的接收者。

最后如图 P.10 所示，消息的发送次序由顺序编号代表。该图中含有两条拥有编号 2（代表步骤 2）的消息，和两条拥有编号 3（代表步骤 3）的消息并非错误。这代表一种并发处理可能性，这两条消息可能被同时发送。在本例中，路由器设置了一个计时器（如图 P.10 的上方所示），并通过并发方式向接收者发送了一条消息（步骤 2），同时向计时器发送了一条开始计时的消息（步骤 2）。路由器可能无法在计时器设置的时限内收到接收者的回复消息，此时计时器会向路由器发送一条超时消息（步骤 3）。如果路由器在计时器设置的时限内收到了接收者的回

复消息（步骤 3），那么客户端就会在步骤 4 中收到积极的确认消息。否则，客户端就会在步骤 4 中收到表明该操作超时的消息。

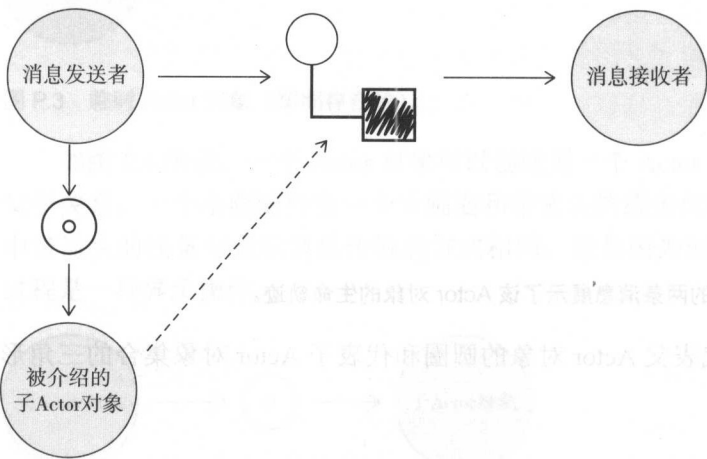


图 P.9 消息发送者通过向消息接收者发送消息，将它的子对象介绍给消息接收者。

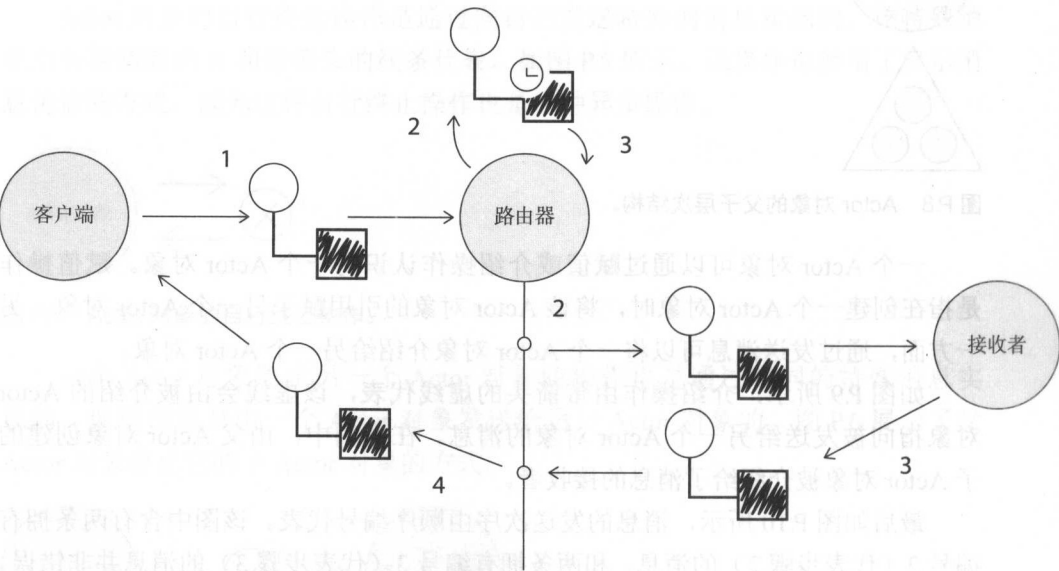


图 P.10 由顺序编号代表的消息发送次序和并发处理情况。

致谢

首先，我要感谢 Addison-Wesley 出版社选择并出版本书。我很荣幸能够再次和编辑 Chris Guzikowski 和 Chris Zahn 一起工作。我特别感谢 Chris Guzikowski，当本书不得不随 Akka 工具集的更改而大规模修改时他给予了极大的耐心。最后，我要说我们的努力是值得的。

如果没有 Carl Hewitt 博士的贡献，一本介绍 Actor 模型的书会变成什么样呢？Carl Hewitt 博士和他的同行们为整个世界贡献了一种独具匠心的计算模型，随着时间的推移该模型会越来越流行。

我还要感谢 Akka 开发团队为 Akka 工具集做出的出色贡献。Jonas Bonér 审阅了本书的多个章节，并作为 Akka 项目的创始人提出了他独特的观点。Akka 项目的技术主管 Roland Kuhn 也审阅了本书的部分章节，并为我提供了宝贵的反馈信息。我要感谢 Roland Kuhn 和 Jamie Allen，他们帮助我建立了本书介绍 Actor 模型的插图规范。此外，我还要感谢 Akka 团队的 Patrik Nordwall，他审阅了本书的前几章。

我特别感谢 Typesafe 公司的 Will Sargent 顾问，他为介绍 Akka 集群功能的章节做了许多贡献。尽管这部分内容很多，但 Will Sargent 从平凡之处找到了闪光点，并帮助我进一步提高它们的品质。

Thomas Lockney 和 Duncan DeVore 是本书两位前期的审稿人。Thomas Lockney 本身就是一位介绍 Akka 框架书籍的作者，Duncan DeVore 在帮助我审阅本书时，自己也正在撰写一本介绍 Akka 框架的书。我特别感谢 Thomas Lockney，当我在本书的前 3 章中进行一些早期尝试时，他给予我许多宽容。坦白地讲，Thomas Lockney 在反复审阅本书和对部分内容进行改进时体现的耐心，令我感到惊讶。

其他为提高本书品质做出贡献的审稿人包括：Idar Borlaug、Brian Dunlap、Tom Janssens、Dan Bergh Johnsson、Tobias Neef、Tom Stockton 和 Daniel Westheide。感谢大家为本书提供的宝贵反馈信息，这些信息使本书能够百尺竿头更进一步。我还特别感谢 Daniel Westheide，他就像一个人体的 Scala 编译器，从本书的示例代码中找出了许多难以发现的错误。

作者简介

Vaughn Vernon 是一位资深的软件开发者，并且是一位简化软件设计和实现思想的领袖人物。他是畅销书 *Implementing Domain-Driven Design* 的作者，这本书也是由 Addison-Wesley 出版社出版的。他还为来自世界各地的数百位软件开发者教授 IDDD Workshop 课程。Vaughn Vernon 经常在计算机行业大会上发表演讲。他擅长的领域包括分布式计算和消息传输，而且尤为擅长 Actor 模型。在 2012 年，他在一个 GIS 系统中第一次使用了 Akka 框架。此后，他就一直专门研究通过由领域驱动的设计模式应用 Actor 模型的技术。通过关注 Vaughn Vernon 的博客 (www.VaughnVernon.co) 和微博 (Twitter 网站的 @VaughnVernon 用户)，可以了解他的最新著作。

10

28

100

101

104

106

107

第 1 章 Actor模型和企业级软件概述.....	1
为什么企业级软件难以开发.....	1
响应式应用程序简介.....	4
响应性.....	5
韧性.....	6
灵活性.....	6
消息驱动.....	7
企业级应用程序.....	8
Actor模型.....	9
Actor 模型的起源.....	10
了解 Actor 模型.....	11
Actor模型的明晰性.....	20
下章提要.....	21
第 2 章 使用Scala语言和Akka框架实现Actor模型.....	22
怎样获取Scala语言和Akka框架.....	23
使用 Typesafe Activator 编辑器.....	23
使用 sbt.....	23
使用 Maven.....	24
使用 Gradle.....	25
使用Scala语言编写程序.....	26
Scala 概要教程.....	27
使用Akka框架编写程序.....	39
Actor 系统.....	40
实现 Actor 对象.....	46
监督.....	52
远程处理.....	55
集群功能.....	68

测试 Actor 对象.....	94
CompletableApp 类	98
小结.....	100
第 3 章 性能情结.....	101
晶体管.....	101
时钟频率.....	103
核心和高速缓存.....	104
可伸缩性.....	106
多线程技术的难点.....	109
Actor模型的作用.....	114
处理伪共享.....	116
设计模式.....	117
第 4 章 通过Actor对象传递消息.....	119
消息通道.....	120
消息.....	121
管道和过滤器.....	126
消息路由器.....	131
消息译码器.....	134
消息端点.....	135
小结.....	137
第 5 章 消息通道.....	138
点对点通道.....	140
发布—订阅通道.....	143
本地事件流.....	143
分布式发布—订阅通道.....	149
数据类型通道.....	157
非法消息通道.....	159
死信通道.....	161
确保送达机制.....	164
通道适配器.....	172
消息桥.....	174
消息总线.....	180
小结.....	189
第 6 章 消息结构.....	190
命令消息.....	191

文档消息.....	192
管理处理流程和处理过程.....	194
事件消息.....	195
请求—回复模式.....	197
返回地址.....	199
相关标识符.....	203
消息序列.....	204
消息有效期.....	206
格式标识符.....	209
小结.....	213
第7章 消息路由.....	214
基于内容的路由器.....	215
消息过滤器.....	219
动态路由器.....	223
接收者列表.....	232
分离器.....	241
聚合器.....	245
重新定序器.....	252
组合消息处理器.....	259
分散—聚集路由器.....	260
传送名单.....	274
处理过程管理器.....	282
消息经纪人路由器.....	298
小结.....	301
第8章 消息转换.....	302
封装器.....	303
内容丰富器.....	305
不可变的 DoctorVisitCompleted 消息.....	309
是否应在本地系统中创建 AccountingEnricherDispatcher 对象.....	309
内容过滤器.....	310
存放证.....	313
标准化器.....	321
规范化消息模型.....	322
Actor 系统需要标准.....	323
小结.....	324

第9章 消息端点.....	325
消息传输网关.....	326
消息传输映射.....	332
事务型客户端/ Actor对象.....	339
事务型客户端.....	341
事务型 Actor 对象.....	342
轮询消费者.....	350
资源轮询.....	354
由事件驱动的消费者.....	358
具有竞争性的消费者.....	359
消息调度器.....	361
选择性消费者.....	364
持久订阅者.....	367
幂等接收者.....	370
避免处理消息副本.....	370
使消息具有相同的效果.....	371
使状态切换操作不受收到消息副本的影响.....	372
服务激活剂.....	378
小结.....	379
第10章 系统管理和基础结构.....	380
控制总线.....	380
改道器.....	382
窃听器.....	384
消息元数据/历史记录.....	385
消息日志/存储器.....	389
智能代理.....	392
测试消息.....	397
通道净化器.....	399
小结.....	401
附录A .NET平台上的Akka工具集: Dotsero.....	402
Dotsero的Actor系统.....	402
通过C#和.NET使用Actor对象.....	405
Dotsero实现.....	410
小结.....	413
参考资料.....	414

第 1 章

Actor模型和企业级软件概述

开发企业级软件的难度很高。

注意，虽然本书主要介绍并发技术，但本书的第一句不是说用到并发和并行技术的多线程软件开发工作的难度非常高。我仅是想告诉大家，开发企业级软件的难度非常高。不过不必过分担心。本书介绍了大量的并发和并行技术，以及提高性能和效率的技巧。你可能对并发技术一无所知，也可能对它们的难度望而生畏。因此，我在本书的第一句话中就指明了并发技术的重要作用。开发企业级软件的难度非常高，远超乎大家的想象。

使用 Actor 模型开发响应式程序，能够使开发工作变得简单并且事半功倍。下面详细介绍 Scala 语言中的 Actor 模型和 Akka 框架，以及它们的性能优势。

为什么企业级软件难以开发

在介绍响应式编程技术会使软件开发工作变得简单之前，让我们先了解一下开发企业级软件为何如此之难。当然，你可能在开发企业级软件时遇到过各种难题，如需要经常使用的数据库出问题、应用程序服务器拖慢了你开发的软件的运行速度，或者更为严重的情况，你使用的编程语言带来了问题。这些问题是普遍存在的。

我们必须面对这些问题，研究出解决方案。请花一些时间仔细思考，围绕数据中心存在的所有物理层、应用程序服务器、软件层、框架和模式、工具集、数据库、消息传递系统和第三方应用程序。这真是一个惊人的结构，如图 1.1 所示。实际上，该图仅是对真实企业级应用程序的高度概括。然而，我们还需要了解更详细的内容。

当然，资深的架构师和开发者仅凭该图就可以了解许多信息。资深的架构师和开发者可以直接使用该图。然而，初中级开发者使用该图会遇到问题，他们会忽视该图省略的信息。

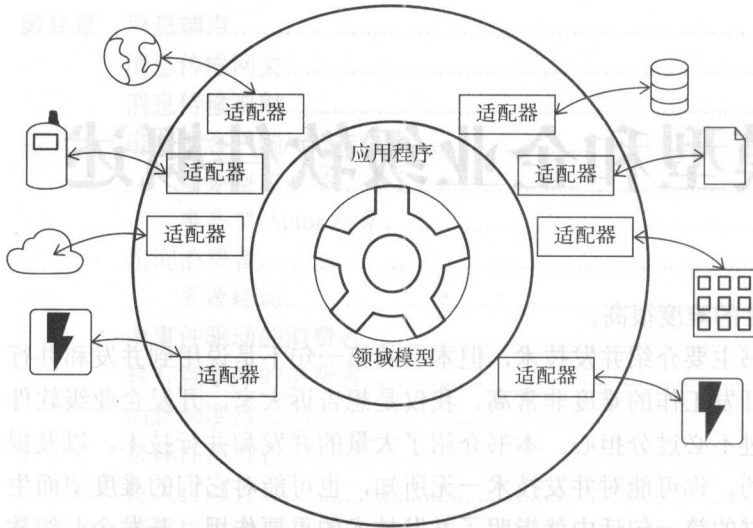


图 1.1 这是一个典型的端口和适配器模式的企业级应用程序架构。它可以是本图所示的圆形，也可以是六边形，但其中包含的层次和复杂组件都是相同的。

在 *Implementing Domain-Driven Design (IDDD)* [IDDD] 一书中，我详细介绍了使用端口和适配器（也称为六边形）架构的原因；这样做可以简化开发企业级软件的工作。至少对于典型的 N 层应用程序来说，这种架构可以简化开发工作。要实现完整的企业级应用程序，必须考虑架构、设计、配置、实现和组件的详细部署工作，如图 1.2 所示。

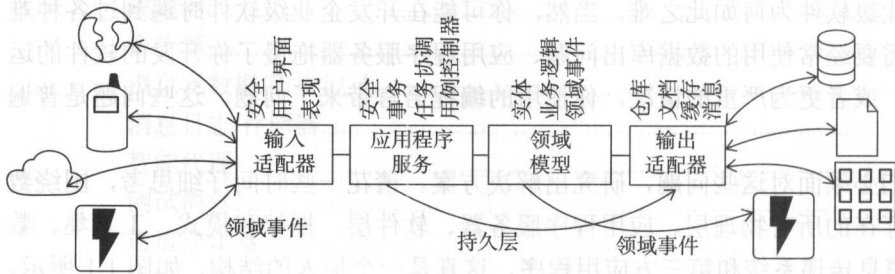


图 1.2 这是一个复杂性堆栈。其中含有许多典型企业级应用程序中跨越多个层面的软件开发要点和必须完成的任务。完成了这些任务后，会使软件开发工作变得简单。

事实上，即使对于大多数资深架构师和开发者来说，这也是一项需要花费数月才能完成的工作，更何况那些初中级开发者呢？将这项工作交给初中级开发者完成，你是否会放心？将这样复杂的工作交给初中级开发者来做显然是有风险的。每位开发者都需要从头至尾了解并熟练掌握的工作的数量是惊人的。我没有

用一个详细的表格来面面俱到地列出企业级软件的复杂性，而是将这些难点总结为一个“复杂性堆栈”。

下面介绍企业级软件开发中的一小部分工作——领域事件 [IDDD]，如图 1.2 所示。持久存储的领域事件代表了业务领域中发生的事情。它们能够传输领域模型中出现过的情况，它们的信息量很大，而且只要你处理领域事件，就需要用到它们。

领域事件仅是一种结果。每个领域事件都是作为在领域中执行的某个命令的输出结果而被创建的。在命令模式 (GoF) [GoF] 中，应用程序中的对象表达了用户或应用程序的目的，从而实现业务操作。一旦该目的被实现，那么获得的结果就是领域事件。图 1.3 展示了一条底线——在任何应用程序中你真正努力实现的，是向领域模型提交命令并获得作为输出结果的领域事件。这种模式非常简单并且具有极为强大的功能。它几乎能够被称为简单性堆栈了。

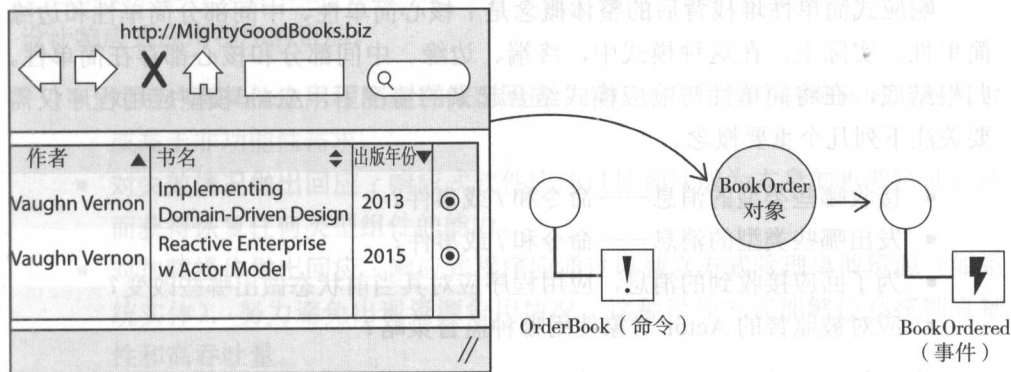


图 1.3 这是简单性堆栈。大幅度精简典型架构后，可以通过设计精良的企业级应用程序获得这种模式。

有人可能会将简单性堆栈和复杂性堆栈混为一谈。事实并非如此，复杂性堆栈中含有随机复杂性和更难处理的固有复杂性。这些复杂性都是客观存在的。简单性堆栈排斥所有无关事物。归根结底，如果你向系统（如浏览器、智能手机、云计算、消息系统等）输入命令，就会获得 Actor 对象和领域事件。因而，你最终实现的是“响应式堆栈”和 Actor 模型。

什么是 Actor 模型？

Actor 模型使 Actor 对象能够像第一类值一样被处理，并通过消息传递功能加强了 Actor 对象之间的通信操作。因为消息是以异步方式发送的，所以对 Actor 对象执行的操作也是高度并发化的，这自然会使应用程序能够以并行方式解决问题。通常，每个 Actor 对象都负责单个的应用程序任务。图 1.3 所示的 BookOrder 就是一个 Actor 对象。

因为 Actor 是一种无锁的并发代理对象,而且执行它们的线程通常不会被阻塞,所以使用 Actor 对象设计的应用程序能够完美地利用系统的基础线程资源。

本补充材料仅是对 Actor 模型的简略介绍。本书通篇会详细介绍 Actor 模型和 Akka 工具集的用法。

响应式简单性堆栈中的安全、事务、持久层、消息等元素具有哪些作用呢?这些元素都代表了应用程序中的真实组件(也许这些元素在解决方案中隐藏得很精妙,你甚至没有注意到它们)。例如,一个 Actor 对象可能既是完美的业务概念实体,又是一种事务一致性边界。另一个例子是使用 Play Framework¹ 工具集,它会令用户界面拥有响应性。正如本书后面内容所述,这些元素无处不在。然而,这种系统级分析模式明显将最重要的元素(业务领域)放在了次要位置。

响应式简单性堆栈背后的整体概念是:核心简单性、中间部分简单性和边缘简单性。实际上,在这种模式中,终端、边缘、中间部分和核心都存在简单性。归根结底,在将简单性与响应模式结合起来的情况下,Actor 模型应用程序仅需要关注下列几个重要概念。

- 接收哪些类型的消息——命令和/或事件?
- 发出哪些类型的消息——命令和/或事件?
- 为了回应接收到的消息,应用程序应对其当前状态做出哪些改变?
- 应对被监督的 Actor 对象使用哪种监督策略?

前两点是应用程序与应用程序的其他组成部分、企业的其他部门和外界的通信方式。第3点描述了应用程序的内部协定,以及 Actor 对象满足业务领域需求的方式。

现在你可以放心地将开发工作交给初中级开发者了,因为这项工作已经被大幅度简化了。他们可以将注意力集中在 Actor 对象协定和测试遵守这些协定的方式上。任何使用 Actor 模型的开发者都能够将注意力集中在具有原子性的单个 Actor 对象的小世界中,轻松地查明它的操作和变化方式。

响应式应用程序简介

响应式应用程序具有较好的响应性、韧性和灵活性,而且应该是由消息驱

¹ Play Framework 是 Typesafe 公司出品的一个响应式用户界面工具集(Scala 语言和 Akka 框架也都是由 Typesafe 公司出品的)。本书没有涵盖用户界面的内容,要详细了解 Play Framework,请参阅本书列出的参考文献。

动的，从而能够使用户获得实时操作的感觉。本节沿用响应式宣言（Reactive Manifesto）[Reactive Manifesto] 的定义。

响应式宣言提出了当前企业与 Web 应用程序（大部分是单线程的）之间的问题。虽然各种计算机宣言的价值有高低，但是响应式宣言确实提供了普通和理想响应式应用程序的完整定义。这有助于在响应式编程思想和当前典型的单线程编程思想之间架起桥梁，将响应式的、事件驱动的编程模式引入软件开发中。而且作为创建响应式应用程序的领先工具之一，Actor 模型的使用指导原则并未受到响应式宣言的限制。

响应式应用程序拥有韧性应用程序堆栈。当被部署到多核系统中时，响应式应用程序可以获得最佳效率。响应式应用程序还经常被部署到多节点集群中，其中包括基于云计算的动态供应环境（Dynamic Provisioning Environment）。下面是设计响应式应用程序的目的。

- **对用户和其他应用程序组件做出回应：**响应式应用程序的响应时间应符合或高于非功能性需求。
- **对失效情况做出回应：**响应式软件应通过使韧性成为本身的重要特征，从而获得恢复任何类型组件的能力。
- **对加载操作做出回应：**响应式程序应通过以独立方式管理离散资源（如系统实体），努力避免出现资源争用情况。这种设计方式能够使系统拥有韧性和高吞吐量。
- **对消息做出回应：**响应式应用程序的基础设计思想是通过其核心的异步消息传递模式，实现消息驱动模式。

对于企业的未来来说，下面是响应式应用程序的关键特性。

响应性

响应式应用程序必须像用户界面的功能需求一样具备响应性。这些功能需求包括实时用户界面的功能需求，实时用户界面允许多个用户同时执行重叠的编辑操作。而且，即使出现故障，响应性也必须仍然存在。因此，响应式应用程序必须兼具响应性和韧性。

通过被观察者和观察者模型可以实现响应性；被观察者和观察者模型是指当系统发生改变时，系统有能力通知对该改变感兴趣的一方或多方。该模型需要使用能够根据用户消耗资源的数量进行调整的事件流和可视化模型，而不是使用根据业务操作调整的系统模型。

韧性

此处的韧性并不是要求开发团队编写出永远不出故障的应用程序。其真正意义是指，应使应用程序拥有故障恢复能力。无须赘言，系统的韧性非常重要。你只需回想一下为了向客户解释上次重大系统停运事故而撰写的系统故障分类和修复报告，就能够理解系统韧性的重要性了。不论这类报告如何精妙华丽，阅读它们的人和撰写它们的人都不会感到愉快。

在 Java 或 .NET 平台上运行的典型多线程应用程序，通常会使用局部化方式处理系统故障。当应用程序出现异常时，维护人员会尝试记录该情况，而且可能会执行一些处理操作。如果在不提升处理等级的情况下，就能够排除这些故障，那么说明这些故障都是小问题。因此，有人认为，既然在各种复杂的故障情况（有些故障甚至难以预测）中难以实现韧性，那么最好的解决方式就是将错误记录下来，让用户使错误最终引发的故障暴露出来。当然得出该结论是有一定道理的，毕竟在难以预测故障的情况下，又怎么能够未雨绸缪呢？

响应式应用程序通过监督较低层级的响应式组件来预测故障情况。这种模式拥有异步操作边界，而且能够将故障具体化为消息，并通过重要的专用消息通道发送 [Read-Write]。监督者会被赋予对被监督组件的故障做出回应的能力。正确的回应可能是彻底停止故障组件，也可能是重新启动故障组件，还可能通过忽略故障原因来命令故障组件继续运行。监督者甚至可以选择使本身失效，从而使它的监督者能够选择上述恢复操作之一。

这种方式倾向于将故障隔离在它们出现的应用程序区域中，从而使程序员能够以对症下药的方式处理它们。同时这还能够保护应用程序的其他组成部分，避免故障以连锁反应的方式影响一个或多个不相关的应用程序区域。

一个关于响应式韧性的结论是，响应式系统有充分的理由被应用到人工智能中。人工智能是指，使软件获得学习和了解新的或未知事物的能力的计算机科学分支。使用监督机制并使响应式组件能够从错误情况中获取信息，可以不断增强应用程序处理未来系统故障的能力。

灵活性

每当我们思考可伸缩性时，总是会横向或纵向扩展思考范围。纵向可伸缩性可以通过添加拥有更多中央处理器（CPU）的高性能计算机实现，每台计算机都拥有多核处理器（如 Intel Xeon Phi 处理器）和大量的内存。横向可伸缩性可以通过添加多台提供日常服务的服务器实现，每台服务器都应拥有中等性能的 CPU（如一块或两块 Intel i7 Quad Core 4700HQ 处理器）。

当然，为了满足特定的可伸缩性需求，也可以同时使用这两种扩展方式。

但从实践的观点看，灵活性比可伸缩性更为重要，因为灵活性还意味着通过调整满足当前应用程序的需求。也就是说，可能需要在非高峰时间通过调整使用较少的计算资源。不论增加计算资源还是减少计算资源，你编写的软件都应该全天候提供与预期相符的响应性。灵活性能够提供这项支持，因为灵活性意味着根据需求进行调整，这种调整方式是响应性的核心。

因为响应式应用程序是由消息驱动的，在这种模式中组件仅会在收到消息时执行操作，所以它们特别适合以更为实用的方式进行调整。此外，响应式组件还可以通过独立于其本身特性的方式，在指定计算节点中执行操作。这种能够包含任何响应式组件的节点，能够通过各种方式被确定和更改。换言之，响应式应用程序会得益于其组件的位置透明性。

尽管位置透明性是用于解决网络划分问题的，但也可以使用它处理在网络中运行的分布式软件。更确切地说，它正好适用于网络。因此，你需要努力平衡位置透明性与认知需求的关系，并将网络映射为编程模型的普通组成部分。

响应式组件的消息驱动特性和它们的位置透明性都为根据需求调整应用程序提供了帮助，即实现了应用程序的灵活性。

消息驱动

因为系统组件仅会在收到消息时做出回应，所以系统能够使用可用线程运行应用程序中必须立刻对消息做出回应的部分。而且，当前没有正在对消息做出回应的组件不会占用宝贵的 CPU 资源。消息的类型包括命令消息、文档消息和事件消息。

因为响应式应用程序中的组件会通过异步消息传递模式，接收其他组件发送的消息，所以能够自然而然地降低各种组件之间的接口和时间耦合性。因为响应式组件能够选择以独立方式对每条消息做出回应的方式，所以它们能够做好接收预期内消息的准备工作。这就进一步降低了发送消息组件和接收消息组件接口的耦合性，因为客户端无须知道发送消息的次序。实际上，从接收消息的次序方面来讲，响应式组件无法获得系统级的保障。

因为响应式组件本身是小型的类似原子的单元，而且它们在同一时刻仅会对一条异步消息做出回应，所以它们能够排除所有锁策略。因为响应式组件无须使用锁策略，所以它们能够将 CPU 资源解放出来，专门处理纯吞吐量操作，并在占用线程时一直保持不断工作的状态。因为组件能够根据需求对消息做出回应（而不是像轮询和阻塞机制那样持续占用 CPU 循环），所以能够提高应用程序的响应性。

企业级应用程序

用于处理企业日常工作的应用程序的种类很多而且涉及的范围也很广。根据不同的业务类型，你能够预测出哪些种类的应用程序一定会被用到。请思考你的公司会使用下面哪些种类的应用程序？

会计、账目（金融等）、航空航天系统设计、自动化交易、网银、预算、商务情报、业务处理、索赔、临床、协作、通信、计算机辅助设计（CAD）、内容/文档管理、客户关系管理、电子健康档案、电子交易、工程、企业资源规划、财务、保健医疗、人力资源管理、身份和访问管理、货品计价、库存、IT和数据中心管理、实验室、生命科学、维修维护、生产制造、医学诊断、网络、订单运输、工资表、药品、出版、运输、项目管理、采购支持、策略管理、风险评估、风险管理、销售预测、调度和预约管理、文字处理、时间管理、认购。

虽然不会使用该列表中的全部软件，但其中的许多软件都是企业日常工作中必须使用的。通常，你需要整合这些软件，本书会介绍帮助你完成这项工作的处理模式。那么，设计这些专业应用程序并在公司执行会怎样呢？最严重的业务失误之一是，将封装的应用程序硬塞进策略解决方案中。其结果充其量是使解决方案有一个边界。

因此，让我们来讨论策略信息系统（SIS）[SIS]。这类系统专门用于凭借软件开发获得业务竞争优势的公司。生成策略的软件可完全由你和你想要实现的业务目标来定制。各行各业（如上面列出的许多行业）都会使用各种版本的企业级应用程序。这类应用程序会被用于设计和制造物美价廉的产品，提供新鲜的、品质更高的服务，或者用于提高业务能力，以应对市场变化和更高的客户要求。

也许你正处在一个企业策略项目的初始阶段。你使用“普通”企业级软件开发工具可能已经获取了一些成果，但为了达到策略应用程序的目标，还必须进一步扩大这些成果。可以考虑使用 Actor 型来满足刚性和之前看起来遥不可及的需求。除了性能和可伸缩性需求外，还需要创建软件模型，以便反映业务预测的心智模型。领域驱动设计（DDD）[DDD] 就是专门用于支持 SIS（企业级软件的核心领域，IDDD）开发的。通过阅读本书，你可以了解这种应用程序设计模式，本书在介绍软件建模方式时融入了基于 DDD 的内容。你会发现通过 Actor 模型可以设计坚韧的、灵活的、由消息驱动的软件模型，从而使你的开发团队中的成员能够更轻松地进行推导。

此外，当你面对新的企业策略挑战时，必定需要整合企业中已经存在的系统。

如前所述，本书还会介绍许多整合方式，它们都会使用 Actor 模型。

开发企业策略解决方案和整合各种企业系统工作中的一个要点是对工具的选择。通常，企业架构师会选择不使任何人失去工作的解决方案。这些解决方案通常是膨胀的、缓慢的、不具备可伸缩性的、无弹性的并且昂贵的。当前这些常见的解决方案已经受到了挑战，而你的目标就是在这类膨胀的解决方案中，寻找精炼的、成本效益高的处理方式。

Actor模型

在 *Dr. Dobbs's Journal* 月刊的一次访谈中，面向对象程序设计先驱、Smalltalk 的发明者之一 Alan Kay 说：“Actor 模型超出我预料之外的是它拥有优秀的面向对象的程序设计功能。” [Kay-DDJ] 在发表这一评论的几年前，Alan Kay 就提出在组件之间传递消息是应该投入主要精力研究的重要编程思想，而不应将其视为对象的属性和内部行为 [Kay-Squeak]。

Alan Kay 对 Actor 模型在软件架构、设计、实现以及组件之间的消息传递方面的作用评价非常高，值得我们仔细研究并进行实践。因此，不应仅将 Actor 模型视为发挥多核计算机潜力的手段，它还是一种重要的软件设计思想。它不仅展示了使用 Smalltalk 语言实现软件的方式，还为开发团队提供了提升系统等级 [Kay-Squeak] 的机会。

另一个发人深省的评价是 Gul Agha 在他撰写的 *Actors: A Model of Concurrent Computation in Distributed Systems* 一书中提出的：“Actor 是比顺序进程和值转换函数系统功能更强大的计算模型。” [Agha, Gul] 这意味着什么呢？与其他系统开发实践方法（如支持并发和并行模式的）相比，顺序和函数方式的能力都更差。正如 Gul Agha 所指出的，Actor 模型不仅可以支持基于 Actor 的系统，还可以在实现细节时使用顺序和函数编程方式。而且，Gul Agha 还提出了下面两个论点 [Agha, Gul]：

- 一个 Actor 对象可以创建其他 Actor 对象。
- 一个顺序进程无法创建其他顺序进程。

这两个论点并非贬低顺序程序的作用。而且也不是说，一个顺序进程无法执行其他顺序进程（数十年的主流开发经验已经证明了这一点）。但是，顺序进程仍旧是静态的，而且只能完成专门化的任务。与此相比，在高动态环境（在这种环境中可以根据需要创建和删除 Actor 对象，甚至可以在运行程序时，根据领域

或操作环境需求更改 Actor 对象的行为) 中, Actor 模型既可以利用顺序编程技巧, 也可以利用函数编程技巧。换言之, 响应式系统不仅拥有并发性和分布性, 而且还拥有弹性、动态性、响应性和韧性。

Gul Agha 将上述观点总结为: “在并行系统中, 某个计算操作在其生命周期中能够被分发的程度是一个需要仔细考虑的要素。通过创建新的 Actor 对象, 可以在计算操作的生命周期中以抽象方式提高它的分布性。” [Agha, Gul] 换言之, 通过创建额外的 Actor 对象可以解决任何特定问题。

Actor 模型的起源

Actor 模型是一种用于处理并发计算的数学模型, 它将 Actor 对象用作并发计算的通用基元。与其他计算模型不同, 发明 Actor 模型的灵感源于物理学理论, 如广义相对论和量子力学 [Actor, Model]。

可以将 Actor 模型视为创建响应式应用程序的手段之一。使用这种具体的响应式软件开发方法, 可以细致地处理响应式应用程序的主要方面: 响应性、韧性、弹性和消息驱动性。它通过消息传递方式实现这四个方面的功能。

有趣的是, 在我撰写本书时, 响应式宣言 (Reactive Manifesto) 仅仅诞生了几个月, 而 Actor 模型却根本不是新的编程技术。早在 1973 年 Actor 模型就已经出现。但为什么数十年来 Actor 模型没有广泛流行, 为什么 Actor 模型最近才名声大噪呢?

Carl Hewitt 博士是 Actor 软件开发模型的发明者, 他为研究该模型投入了数十年时间。Carl Hewitt 博士发现某些计算问题需要使用并发和分布模式解决, 但他在做这项研究时第一台晶体管计算机仅出现了 14 年,² 还不存在多核处理器。在 1973 年, 功能最强大的 Intel 处理器仅含有 4,000 至 5,000 个晶体管, 其主频还没有超过 1MHz。图 1.4 展示了 Carl Hewitt 博士在研究 Actor 模型时, 能够使用的功能最强大的处理器。处理器的硬件能力无法将 Carl Hewitt 提出的理论付诸实践。因此, 当时分布式和并行系统知识的理论模型与现实情况不匹配 [Mackey]。直到最近, 才有极少数的应用程序使用了 Actor 模型。

幸运的是, 现在我们已经可以使用 IBM 的 zEC12 和 Intel 的 Xeon Phi 等功能强大的处理器 (当然次一级的 i7 功能也非常强大), 硬件情况已经有了翻天覆地的变化。当今的计算机架构已经有能力支持这个具有 40 多年历史的理论, 这门技术又怎能不枯木逢春呢?

² 参见第3章。

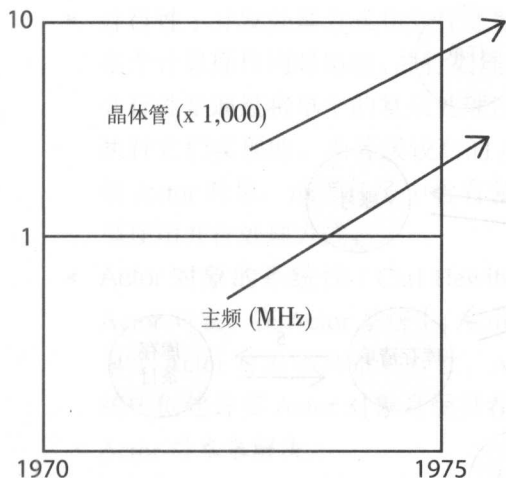


图 1.4 1973 年左右功能最强大的处理器。

了解 Actor 模型

Actor 是一种计算实体，它会对收到的消息做出回应，并且可以做下列事情：

- 向其他 Actor 对象发送一定数量的消息。
- 创建一定数量的新 Actor 对象。
- 设定对下一条消息做出的回应方式。

执行这些操作的次序不分先后，而且可以通过并行方式执行它们 [Actor Model]。

在功能齐全的 Actor 系统中，所有事物都是 Actor 对象。这意味着我们通常使用的基本数据类型（如字符串和整型）都是 Actor 对象。实际上，在实用的现代 Actor 编程语言出现前，将 Actor 对象应用到这种程度会使问题大幅度复杂化。

但即使这样也是可以接受的，而且最实用的模式是设计一种将 Actor 对象用作特殊类型系统组件的 Actor 系统。在这种系统中，Actor 对象的尺寸比整型值大，但也不会比整型值的尺寸超出太多。要确定 Actor 对象的适当尺寸，可将它们视为专门化的应用服务，如单个的领域模型实体或小型的领域模型集合 [IDDD]。虽然这是一种过度简化的描述，但这为大多数企业级软件开发者理解这个概念提供了一个通用支点。当然，Actor 模型的作用并不仅限于此，但不论它们还具有哪些功能，基本上都会遵守单一功能原则（SRP）[SPR]。图 1.5 展示了多种 Actor 对象示例。

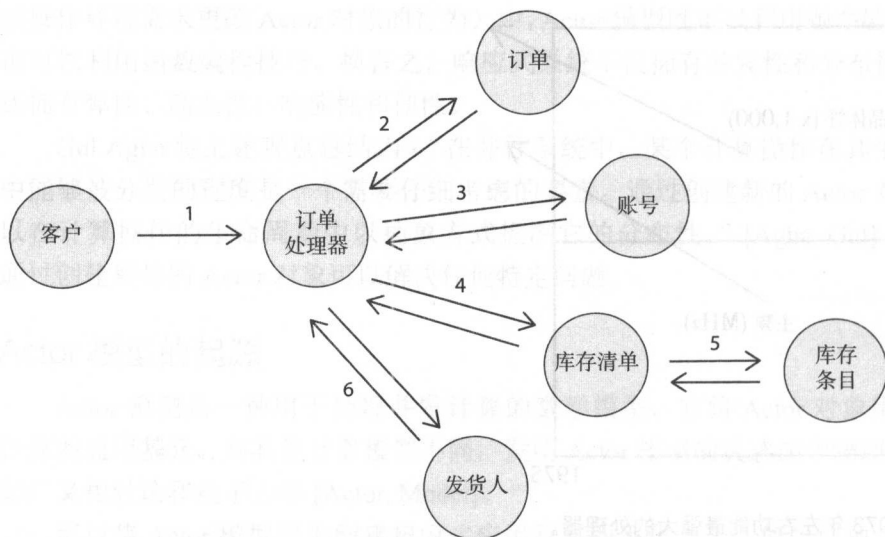


图 1.5 在订单处理业务中可以使用的 Actor 对象的尺寸和功能。

Actor 系统和 Actor 对象具有下列基本特点。

- **直接通过异步消息传递方式进行通信**：如果 Actor 对象 A1 要向 Actor 对象 A2 发送消息 M1，那么 Actor 对象 A1 就必须知道 Actor 对象 A2 的地址。如果 Actor 对象 A1 知道 Actor 对象 A2 的地址，那么它能够直接向 Actor 对象 A2 发送消息 M1，但 Actor 对象 A2 会使用独立线程接收和处理消息 M1。换言之，消息 M1 是通过异步方式被发送给 Actor 对象 A2 的。实际上，在发送者 Actor 对象和接收者 Actor 对象之间还存在一个间接处理层——邮箱（消息缓存单元）。即便如此，我们还是将这种消息传输方式称为直接传输方式，因为编程模型提供了一种宝贵的抽象，该抽象使消息就像直接从一个 Actor 对象传输到另一个 Actor 对象一样。
- **状态机**：Actor 模型支持有限状态机。当 Actor 对象转换为某个预设状态时，就能够改变对未来接收到的信息的处理模式。通过变为另一种消息处理器，Actor 对象就成了一种有限状态机。
- **无共享**：一个 Actor 对象不会与其他 Actor 对象或相关组件共享可变状态。
- **无锁的并发处理方式**：因为 Actor 对象不会共享它们的可变状态，而且它们在同一时刻仅会接收一条消息，所以在对消息做出回应前，Actor 对象永远都不需要尝试锁定它们的状态。因为无须使用锁策略，所以它们能够将多核 CPU 从锁定问题中解放出来，集中精力提高吞吐量，并且使所有处理响应式组件的线程不被阻塞。

- **并行性**：并发处理方式和并行处理方式是不同的概念。并发处理方式是指多个计算操作同时出现。并行处理方式是指以并发处理方式完成单个目标。并行性是通过将单个的复杂处理过程拆分成较小的任务并以并发处理方式执行它们实现的。当等级较高的 Actor 对象能够将多个任务分派给多个下级 Actor 对象，或者任务中含有复杂的处理层级时，就适合通过 Actor 模型使用并行处理方式。
- **Actor 对象的系统性**：Carl Hewitt 博士说：“单独存在的 Actor 对象不是 Actor 对象。在 Actor 系统中，Actor 对象才能成为 Actor 对象。”这就是说，单个 Actor 对象不具备并行性。Actor 对象的量级非常轻，因此在单个系统中创建许多 Actor 对象是受推荐的处理方式。任何问题都可以通过添加 Actor 对象来解决。

Actor 模型除了具有这些基本特点外，还有一些扩展特点。下面是 Akka 框架具有的特点。

- **位置透明性**：使用抽象引用代表 Actor 对象的地址。如果 Actor 对象 A1 获得了 Actor 对象 A2 的引用，Actor 对象 A1 就能够向 Actor 对象 A2 发送消息。提供支持的 Actor 系统会负责处理传送消息的操作，不论 Actor 对象 A2 是位于本地 Actor 系统还是位于远程 Actor 系统中。
- **监督**：在 Actor 对象之间建立依赖关系，父 Actor 对象监督子（下级）Actor 对象。当监督者 Actor 对象向下级 Actor 对象分派任务时，就必须对这些下级 Actor 对象出现的失效情况做出回应。合法的回应包括继续运行、重启和停止下级 Actor 对象。监督者还可以通过使本身失效从而使失效情况升级，这会将失效控制权上交给监督者的父对象（监督者的监督者）。监督机制适于在并行处理方式中使用，在该方式中监督者会将多个任务分派给多个下级对象，从而形成任务处理层级。
- **Future/Promise 对象**：这两种对象提供了接收异步操作结果的手段，不论该结果是代表异步操作成功完成还是异步操作出现失效情况。为了管理接收到的结果，系统需要使用特殊的 Actor 对象（如 Future 和 Promise）。拥有 Future 对象的组件可以选择以等待 / 阻塞方式接收结果，也可以选择以异步方式接收结果。

作为一种计算实体，Actor 对象与原子类似。在有可用线程的情况下，每个 Actor 对象都会在收到消息时处理这条消息并且在同一时刻仅能处理一条消息，如图 1.6 所示。Actor 对象的性能完全由其本身的吞吐量决定，因此可以说，Actor 对象是按照自己的节奏在工作。然而，因为 Actor 对象不会与其他 Actor 对

象共享可变状态，所以当通过计算或数据处理操作处理收到的消息时，Actor 对象也不必锁定系统资源。

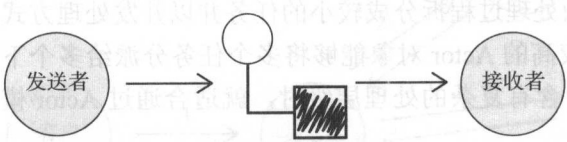


图 1.6 在有可用线程的情况下，每个 Actor 对象都会在收到消息时处理这条消息并且在同一时刻仅能处理一条消息。

在无锁情况下，理论上，Actor 对象的吞吐速度会非常快。即使在某些特殊情况下，如果计算和处理操作比较小、集中和耗时较短，那么通常消息就会以极为迅速的方式被发送、接收和处理。这就使你有可能避免使用阻塞机制和导致阻塞问题的机制（串行设备）。³ 因此，Actor 系统就能够拥有高性能、高吞吐量和低操作延迟。

对于基于 Actor 的响应式系统来说，并发和并行处理方式极为重要。根据阿姆达尔定律（Amdahl's law），通过使用多核处理器和并行处理方式运行程序而获得的性能提升，受限于执行程序中顺序部分所需的时间。例如，如果使用单核处理器运行某个程序需要 20 个小时，而该程序中有 1 个小时的部分无法以并行方式执行，另外 19 个小时（95%）的部分能够以并行方式执行，那么在执行该程序时不论增加多少个处理器，执行该程序所用的最少时间也无法降至 1 小时以下。在本例中，性能提升被限制为至多不会超过 20 倍，如图 1.7 所示。

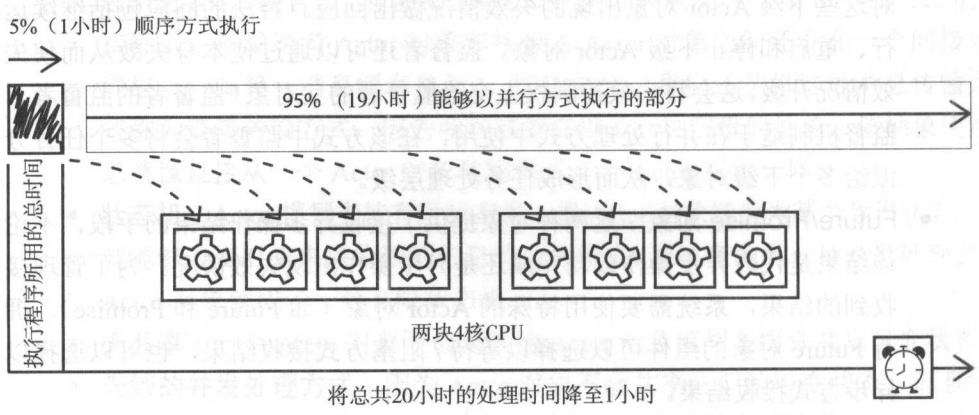


图 1.7 阿姆达尔定律表明，系统性能的最大提升幅度由无法并行化的部分决定。

³ 显然，要在大多数企业级应用程序中彻底避免使用串行设备（如硬盘）是不可能的。在这类情况中，我会尝试尽量在非阻塞抽象中实现必要的依赖关系或者受限的阻塞访问操作。要获得该问题的详细解决方案，请参阅第9章。

无法并行化的 1 小时部分由哪些操作组成呢？这部分程序中可能仅包含对各种设备的输入/输出（I/O）阻塞操作。然而，几乎所有程序中都会含有这类部分，而且如果通过异步方式处理这类部分，就会增加系统的复杂性。

阿姆达尔定律着重指出了设计并发系统的重要性。要设计并发系统，必须根据实际情况最大限度地利用最大数量的 CPU 核心。请注意我特意使用的“实际情况”一词。特定系统的特定区域有可能会受到现实情况的限制，过高的并行化程度会提高系统的复杂度，超出你的开发团队的推导能力。

例如，通过并行化可以提高图 1.7 所示系统 5% 部分的吞吐量和性能。如前所述，为了实现这一小部分的性能提升会改变 I/O 阻塞操作。然而，如果为了获得这一小部分的性能提升而增加系统的复杂性，进而导致超出你推导系统的能力，那么这样做就得不偿失。通常，应从大处着眼寻找能够获得性能大幅度提升的其他区域，而不应死盯仅能提供系统很小提升的某个小部分。

Actor 对象在收到一条消息后能够切换到新状态，但在 Actor 对象彻底完成状态切换前，新状态是由中间计算操作或其他处理操作决定的。然而，该 Actor 对象在这些操作被执行时必须保持回应性。怎样才能做到这一点呢？可使用请求—响应模式。当一个 Actor 对象向另一个 Actor 对象发送异步消息时，它就初始化了一个最终回应协定。因为执行初始化协定操作的 Actor 对象不会因等待回应而被阻塞，所以它仍然可以处理后续消息，直到收到回应为止。请求—响应是在 Actor 系统中广泛使用的基础模式。

如果 Actor 对象必须在收到回应消息后才能继续处理其他消息，会出现怎样的情况呢？因为 Actor 对象必须仍旧保持回应性，所以在收到回应前，Actor 对象会处于暂停状态并将收到的所有新消息都存储在内部缓冲区中⁴。一旦收到预期的回应，Actor 对象就会像往常一样处理所有被缓存的消息。但是，如果 Actor 对象需要花很长时间才能收到回应，而且缓存消息的数量无限制地增长，应该怎么办呢？那么处于暂停状态的 Actor 对象就必须对超出其缓存容量的消息发送者，发出“抱歉，我现在无法处理工作，请稍后再试”之类的消息。

此外，Actor 系统本身无须关心 Actor 对象之间发送哪些类型的消息。接收消息的 Actor 对象会根据其本身的状态，以正确方式做出回应。当然，客户端也必须了解 Actor 消息的协定，并能够发送其支持的消息。

从状态切换方面看，当 Actor 对象处理收到的消息时，它就能够为处理将来收到的消息做准备。简言之，Actor 对象可以通过使用另一种操作替换当前收到

⁴ 缓存消息不是 Actor 模型的自带功能，但是 Akka 框架提供了该功能。也可以根据自己的喜好，使用其他基于 Actor 模型的工具缓存消息。

的操作。在内部, Actor 对象可以将当前收到的操作彻底替换为另一种操作, 它也可以将先前收到的操作存储到堆栈中, 然后再将它们弹出堆栈并使用。⁵ 一个 Actor 对象可能会收到多个不同类型的操作, 可以在这些操作之间切换, 但它缓存操作的数量是有限的。这个特性使 Actor 对象能够成为高效的有限状态机。

显然, 有些情况比较适于使用 Actor 对象的操作切换功能, 而有些情况不适合使用该功能。因此不应滥用这一功能。例如, 如果 Actor 对象只有在收到指定的消息后才能切换到某个特定状态, 那么当它收到的消息队列中含有指定的消息时, 就可以假设该 Actor 对象在其当前实例的生命周期中已经收到了这条指定的消息。这种假设会对处理问题提供帮助。

在讨论 Actor 模型时, 我不会介绍消息传递功能的普通使用方式。Actor 对象的消息传递功能, 与发布-订阅模式 [POSA1]、消息队列和基于 SOAP 请求的远程过程调用 (RPC) 中的消息传递功能不同。当然, 你可能会说这三种处理方式也是 Actor 模型, 或者说至少它们是旧式的 Actor 模型。实际上, Carl Hewitt 甚至说过可以使用 Actor 模型创建电子邮件系统和 Web 服务端点 [Hewitt-Actorcomp]。

另一方面, Carl Hewitt 博士和他的团队还介绍了一些工具, 这些工具可以帮助你更好地使用 Actor 模型。这是我想着重介绍的 Actor 模型的用法, 而且我使用的 Akka 框架就是这些工具之一。

管理不确定性系统

什么是不确定性, 为什么我们要关心它呢? 在应用程序开发过程中, 不确定性系统是指当使用相同的输入数据多次执行程序时, 会输出不同结果的系统。例如, 你使用某个应用程序处理数据 A, 第一次执行该程序时, 该程序的输出结果为 B, 而第二次执行该程序时, 该程序的输出结果为 C。也就是说, 在使用该应用程序处理数据 A 时, 你无法预测会得到结果 B 还是结果 C, 因此该程序具有不确定性。不确定性系统是否就意味着不可靠呢? 答案是否定的。而且, 由事件驱动的响应式应用程序天生就具有不确定性。当然, 该特点不会增加人们对响应式应用程序的信任度。

实际上, 不应将不确定性和不可靠混为一谈, 只要你了解了应用程序产生不确定输出结果的情况, 就不必过于担心程序的不确定性。下面介绍一个不确定性系统示例。

示例描述: Actor 对象 Customer (代表客户) 要求 Actor 对象 OrderProcessor (代表订单处理器), 在订货簿 B 中创建了一个 Actor 对象 Order (代

⁵ 在 Akka 框架中, 该功能称为变化 (become), 该功能使 Actor 对象能够变化成另一种消息接收器。

表订单), 成功创建了该对象后, 应命令 Actor 对象 Account (代表账户) 处理 ChargeCreditCard 消息。一旦处理完 ChargeCreditCard 消息后, Order 对象就会拥有输出信息的引用。ChargeCreditCard 请求是有时间限制的。Customer、OrderProcessor、Order 和 Account 对象必须通过响应方式创建。

1. Customer 对象命令 OrderProcessor 对象为订货簿 B 创建一个 Order 对象。
2. OrderProcessor 对象为订货簿 B 创建了 Order 对象后, 为表明该事实, 事件 OrderPlaced 就会被创建。
3. 事件 OrderPlaced 会被从 Order 对象传送到 OrderProcessor 对象。
4. OrderProcessor 对象命令 Account 对象处理 ChargeCreditCard 消息, 成功处理该消息后, 为表明该事实, 事件 AccountCharged 会被创建。
5. 事件 AccountCharged 会被从 Account 对象传送到 OrderProcessor 对象, 从而命令 Order 对象存储由 AccountCharged 事件传送的信息的引用。
6. 通过命令 Inventory 对象 (代表存货清单) 保存订货簿 B, 并命令 Shipper 对象传送 Order 对象, 示例程序会继续运行。

这是令人满意的处理方式, 因为最终这个不确定性系统生成了预期的输出结果。但是, 该系统可能会遇到各种各样的情况, 而且会生成不同的输出结果。例如, OrderProcessor 对象会由于某些原因而失效, 无法创建 Order 对象, 甚至在纯确定性系统中这种失效情况也会出现。因此, 既然这种错误情况不是我们关心的重点, 那么此处的要点是什么呢?

使整个系统具有不确定性的关键点之一, 是创建 AccountCharged 事件的操作必定有时间限制。因为在创建 OrderPlaced 事件的操作和创建 AccountCharged 事件的操作 (实际上是传送 AccountCharged 事件的操作) 之间存在超时情况, 所以在处理过程 A 中会有两个潜在的输出结果:

- B. AccountCharged 事件及时被 OrderProcessor 对象收到, Order 对象被命令存储 AccountCharged 事件传送的信息的引用。
- C. Account 对象延迟了创建 AccountCharged 事件的操作, 或者在传输 AccountCharged 事件时出现了网络延迟 / 故障情况, 因而尽管 Order 对象已经被创建, 但它还没有获得 AccountCharged 事件传输的信息的引用。

如果处理过程 A 的输出结果是我们期望的输出结果 B, 那就万事大吉了。但是,

如果输出结果是 C 又会怎样呢？这是否意味着该系统出了故障、不可靠甚至有严重缺陷呢？事实并非如此。这仅仅意味着即使你特别想要获得输出结果 B，但还是需要处理输出结果为 C 的情况。怎样处理这种情况呢？这需要编写新的程序。

示例描述：OrderProcessor 对象命令 Order 对象重新发送获取 Charge-CreditCard 消息的请求。

1. 从处理周期方面看，OrderProcessor 对象会命令 Account 对象不断请求获取 ChargeCreditCard 消息，直到成功获取该消息或者到达操作时限并再次尝试执行该操作为止（如果到达了操作时限，OrderProcessor 对象会命令 Order 对象发送 Cancel 消息，代表取消当前的请求操作）。
2. 当成功获得 ChargeCreditCard 消息（该消息可能之前就已经成功被发出，但因为网络延迟 / 故障而没有被收到）后，Account 对象就会创建事件 AccountCharged。
- 事件 AccountCharged 会被从 Account 对象传送到 OrderProcessor 对象，从而命令 Order 对象存储由 AccountCharged 事件传送的信息的引用。
3. 通过命令 Inventory 对象（代表存货清单）保存订货簿 B，并命令 Shipper 对象传送 Order 对象，示例程序会继续运行。

整个系统具有不确定性的现实情况，要求你思考完整的业务处理过程和最终必须实现的目标。只要你能够在应用程序中识别出并了解生成不确定输出结果的所有情况，就能根据用户和系统所需要的最终输出结果设计成功的应用程序。设计程序的主要关键点是明确软件模型，因为只有这样做才能使你理解复杂的系统。

有人批评 Actor 模型与生俱来的不确定性 [Actors-Nondeterministic]。该评论并不符合事实。实际上，单个 Actor 对象本身具有原子单元安全的确定性。从某种意义上说，整个 Actor 系统在任意时间点都是不确定的，因为以并发方式执行的业务系统在任意时间点都是不确定的。无限的不确定性是围绕 Actor 模型基础理论的早期争论焦点 [Actors-Controversy]。然而，对这些争论总结归纳后，你会发现具有天生不确定性的是由事件驱动的架构，而 Actor 模型只不过恰好是一种由事件驱动的架构。而且，通过引入 Actor 对象（Actor 对象本身是一种具有原子确定性的单元），Actor 模型还能够帮助我们推导天生具有不确定性的并发业务系统。

不论通过怎样的方式实现由消息驱动的系统（也称为由事件驱动的系统），这些系统都是编写具有响应性、韧性和弹性应用程序的基础。而且，响应式系统不必具有高度的不确定性。不可变状态可以为应用程序提供确定性和数据流并发性（使用单一赋值变量）。

因此，真正的决策点是使用无法伸缩的单线程架构设计程序，还是使用可管理的、多线程的、由事件驱动的架构设计程序。随着由事件驱动的架构 [EDA-Verification] 不断增多，有人认为 Actor 模型是处理不确定性的最佳方式。实际上，使用 Actor 模型可以帮助开发团队成功推导复杂的、不确定的系统。

对象性能模型

尽管 Actor 模型与对象模型不同，但是这两种模型的重合区域还存在一种名为对象性能模型 [OCM] 的安全模型。因为在重合的区域中这两种模型的概念重合度较高，所以在下面的内容中我使用“Actor 对象”一词替换了“对象”一词。

Actor 对象只能根据发送消息才能交互。可通过下列方式获得引用。

1. 初始情况：Actor 对象 A 可能一开始就已经拥有 Actor 对象 B 的引用。
2. 父子关系：当 Actor 对象 A 创建了 Actor 对象 B 后，Actor 对象 A 立刻就获得了 Actor 对象 B 的唯一引用。
3. 赠予：当 Actor 对象 A 创建了 Actor 对象 B 后，Actor 对象 A 可以将 Actor 对象 B 的引用赠予其他 Actor 对象。
4. 介绍：如果 Actor 对象 A 拥有 Actor 对象 B 和 Actor 对象 C 的引用，那么 Actor 对象 A 可以通过发送消息使 Actor 对象 B 获得 Actor 对象 C 的引用。Actor 对象 B 可以保留该引用并在将来使用该引用。

这个重合区域很明显。父子关系法则也许是最强的安全模式。当父 Actor 对象创建子 Actor 对象时，在创建操作完成的那一刻，父 Actor 对象拥有子 Actor 对象的唯一引用。这种情况确保了父 Actor 对象能够彻底隔离所有对子 Actor 对象发送的消息，从而使子 Actor 对象获得了它能够达到的最高安全性。除非专门将该子 Actor 对象设计为能够创建它本身的子 Actor 对象，或者能够接收其他 Actor 对象的消息或向其他 Actor 对象发送消息（这会符合对象性能模型的其他法则⁶），否则其他 Actor 对象都无法向它发送消息。

尽管 Actor 模型通常会被用作对象性能模型，但通过这种方式使用 Akka 框架还有一个障碍。Akka 框架支持通过 ActorSelection 功能向 Actor 对象发送消息（请参阅第 2 章）。在使用 ActorSelection 功能的情况下，可以通过路径和名称找到系统中任何由用户管理的 Actor 对象，从而使你能够绕过封装。如果你使用了 ActorSelection 功能，就应该认识到你编写的程序不是与对象性能模型兼容的。尽管如此，理解了对象性能模型还是能够帮助你了解，将 Actor 对

⁶ 请参阅第6章。

象展示给其他 Actor 对象和将 Actor 对象与其他 Actor 对象隔离的方式。

Actor模型的明晰性

在继续介绍下面的内容之前，我要说明本章介绍的主要概念之一，利用 Actor 模型可以大幅度简化企业级应用程序。领域驱动设计的主要目标之一，是在软件模型 [IDDD] 中使业务概念变得明确和清晰。如图 1.1 和图 1.2 所示，所有额外的活动部分和静态配置都无法帮助软件模型获得明晰性。实际上，获取确切业务信息的最佳位置就是图 1.1 中的中心区域——领域模型。然而，架构中含有非常多的活动部分，有时这些活动部分会使你真正想要获得的软件模型蓝图变模糊。

另一方面，如图 1.3 所示，精简的架构仅含有用户界面和清晰的软件模型。据此甚至能够推断出用户界面也是软件模型的组成部分，因为它反映和表现了业务专家的心智模型，通过这种方式帮助用户做出重要决策，并使用软件模型采取重大行动。这就是你应该努力通过 Actor 模型实现的目标：明晰性。

图 1.8 进一步突出了软件模型具有明晰性的重要性。指定消息的接收者是明确的，不论这条消息是命令消息还是事件消息。你只需阅读 Actor 对象的源代码并了解它们的协定（既包括内部的也包括外部的）。不要受到错误观点的干扰，例如有人说 Actor 模型中的 Actor 对象必须紧密耦合。上一节的内容说明这一观点并不正确。图 1.8 所示的 Actor 对象，能够通过赠予或介绍方式轻松获得其他 Actor 对象的引用。如果应用程序的处理过程比较复杂，可以在 BookOrderController 对象（代表图书订单控制器）、BookOrder 对象（代表图书订单）和 OrderFulfillment 对象（代表订单执行结果）之间，添加进程管理器和各种消息路由器。你还可以参考图 1.5 所示的架构。

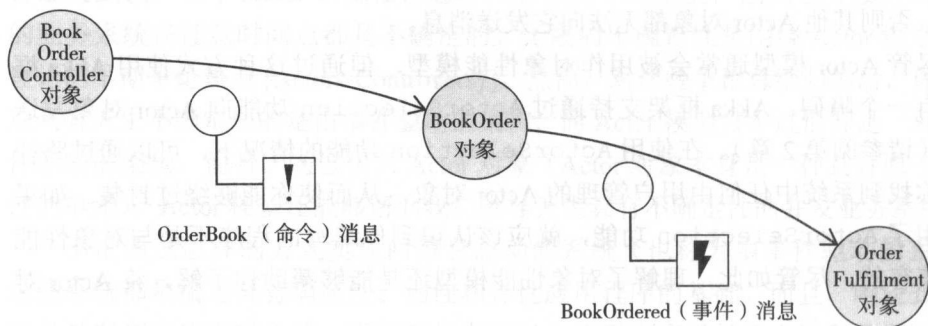


图 1.8 Actor 模型会帮助你抛去其他干扰的观点和思想。你应该更进一步获取 Actor 模型的明晰性。

你应该抛去其他干扰的观点和思想，仅将注意力集中在软件模型和用户界面上。你应该更进一步获取 Actor 模型的明晰性。本书后面的内容将帮助你抛弃过时的编程习惯。你现在已经进入了 Actor 模型及其并发和并行性的快车道。

下章提要

学习了基础知识后，你就可以学习 Actor 模型的用法了。如果你是一位 Java 开发者，能够更加容易地掌握 Scala 语言和 Akka 框架。第 2 章既介绍了 Scala 编程语言，也介绍了 Akka 工具。之后，第 3 章会介绍提高单计算节点和多计算节点性能的方式。

如果你是一位 C# .NET 开发者，我建议你参阅附录 A 的内容，附录 A 介绍了 Dotsero 工具，它是用于 .NET 开发的类似于 Akka 框架的软件库。Dotsero 库会为 C# 开发者提供支持，帮助他们在自己熟悉的平台上使用 Actor 模型。

第 2 章

使用Scala语言和Akka框架实现Actor模型

通过几种独特的方式可以实现 Actor 模型。可以使用 Actor 库工具实现 Actor 模型，也可以直接使用编程语言实现 Actor 模型。ActorScript 是典型的 Actor 编程语言之一。Smalltalk 语言中也含有 Actor 元素。Erlang 语言含有 Actor 元素，但它不是完全基于 Actor 模型的。Erlang 语言支持将 Actor 对象 / 进程作为第一类并发抽象处理。最近，Scala 和 Scratch 等编程语言也支持基于 Actor 的编程方式。

Scala 是一种 Java 虚拟机 (JVM) 语言，其本身不含有 Actor 元素。但是，Scala 语言自带了一个基于 Actor 的工具——Akka 框架。因此，Scala 语言与 Erlang 语言有很多相似的地方，但也有很多不同之处。例如，Erlang 是一种函数式编程语言，其中的所有数据都是不可变的。另一方面，Scala 是一种混合型语言，它将面向对象编程方式和函数编程方式融合到了一起。在使用 Scala 语言时，可以通过可变方式实现某些概念，同时通过不可变方式实现另一些概念。

可以通过 Java 程序使用 Akka 框架

即使你想要使用 Akka 框架，也不必专门学习 Scala 语言。Akka 工具包中含有兼容 Java 的应用编程接口 (API)。但无论如何，通过 Java 程序使用 Akka 框架，不会像通过 Scala 程序使用 Akka 框架那样得心应手。许多人认为 Scala 将会成为一门伟大的编程语言，因为一行 Scala 可以代替数十行 Java 代码。但是，一些公司仍旧使用 Java 语言，而没有投入 Scala 语言的怀抱。因此，通过 Java 程序使用 Akka 框架也是很正常的。然而，你可能会最终使你的公司逐渐改变对 Scala 语言的接受程度。那将会是一个皆大欢喜的局面。

与 Carl Hewitt 博士和他的开发团队最初指定的软件库相比，某些软件库可以使你与 Actor 模型走得更近。然而，开发 Akka 框架的目的并不是要创建一种 Actor 语言。实际上，Akka 软件库 / 框架替代了 Scala 语言默认的 Actor 软件库。尽管你也可以使用 Scala 语言默认的 Actor 软件库，但该库的功能非常有限。此外，scala.actors 库被 Scala 2.11 弃用了，因此 Akka 框架取代了 Scala 简单的 Actor 软

件库后，没有人感到不适应。

如果你了解一点 Erlang 编程语言，就能够轻松地分辨出 Erlang 语言的哪些部分在很大程度上对 Scala 语言和 Akka 框架产生了影响。例如，Erlang 语言的模式匹配功能就作为默认的语言功能出现在 Scala 语言中，并且也是 Akka 框架中的 Actor 功能。

通过上述内容简单了解了 Scala 语言和 Akka 框架后，下面让我们学习如何使用它们。

怎样获取Scala语言和Akka框架

可以通过多种方式获取 Scala 语言和 Akka 工具包。本节介绍使用 Typesafe Activator 编辑器或者 sbt、Maven 或 Gradle 等构建工具，通过代码仓库编写程序的方式。你也可以根据自己的喜好直接下载这些工具。

使用 Typesafe Activator 编辑器

访问 <http://typesafe.com/activator/> 或者 <http://typesafe.com>，可以下载 Typesafe Activator 编辑器。Typesafe Activator 编辑器提供了一个开发环境，它会成为你使用 Typesafe Reactive Platform 的阶梯。使用基于向导的用户界面，可以创建种类繁多的样本项目。创建完指定的项目后，你就可以安全地退出 Typesafe Activator 环境，然后在自己喜欢的开发环境中打开样本项目。

使用 sbt

sbt 是简单构建工具（Simple Build Tool）的缩写词，而且实际上已经在某种程度上成为构建 Scala 程序的标准工具。许多 Scala 开发者使用 sbt，而且他们会同样使用 Akka 框架构建项目。下面列出了为了创建 Scala 和 Akka 项目，需要在定义文件 build.sbt 中添加的代码：

```
name := "RiskRover"

version := "1.0"

scalaVersion := "2.10.4"

resolvers += "Typesafe Repository" at
  "http://repo.typesafe.com/typesafe/releases/"

libraryDependencies += "com.typesafe.akka" %% "akka-actor" % "2.3.4"
```

此处的 Akka 依赖关系设置的是特定的 Akka 版本。你需要确定自己使用哪个版本的 Akka 框架。如果你是第一次使用 sbt，就会发现它不仅有趣而且还拥有强大的功能。虽然 sbt 具有一定的复杂性，但这最终会帮助你提高水平 [Westheide]。如果你想详细了解 sbt，请参阅 Josh Suereth 撰写的 *sbt in Action* [sbt-Suereth]。需要注意的一个要点是，sbt 不仅仅是一个构建工具，实际上它还是一个功能强大的 REPL¹ 编程环境。

使用 Maven

实际上，Maven 已经成为 Java 开发者使用的标准工具，而且也可以使用它创建 Scala 项目。如果你熟悉 Maven，就可以毫无疑问地将下面的声明添加到构建脚本中，然后就可以使你创建的项目中含有 Scala 代码和 Akka 框架：

```
...
<pluginManagement>
  <plugins>
    <plugin>
      <groupId>net.alchim31.maven</groupId>
      <artifactId>scala-maven-plugin</artifactId>
      <version>3.1.6</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>testCompile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</pluginManagement>
...

<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-actor_2.10</artifactId>
  <version>2.3.4</version>
</dependency>
```

¹ REPL是交互式编程环境的缩写词。它是一种高端的Shell界面，使你能够在不创建项目和执行程序的情况下，试验编程语言的功能。

当然，这些声明专门应用于特定版本的 Scala 语言和 Akka 框架。你需要确定自己使用哪个版本。

使用 Gradle

我非常喜欢使用 Gradle，这在很大程度上是因为我不喜欢使用构建模块和构建工具，而且 Gradle 易于掌握并且效率较高。它可能在 Scala 开发者中不太流行，但我认为这主要是 Scala 开发者们对 Gradle 还有误解。我不认为 Gradle 会取代 sbt 成为流行的 Scala/Akka 构建工具，但我觉得它会取代 Maven。使用下列简单的脚本可以通过 Gradle 构建 Scala 和 Akka 项目：

```
apply plugin:'scala'

repositories {
    mavenCentral()

    // 是否使用快照；否则删除快照；
    // 查看 akka 框架的依赖关系，
    maven {
        url "http://repo.akka.io/snapshots/"
    }
}

dependencies {
    // 设置指定的版本
    compile 'org.scala-lang:scala-library:2.10.4'
}

tasks.withType(ScalaCompile) {
    scalaCompileOptions.useAnt = false
}

dependencies {
    // 设置指定的版本
    compile group: 'com.typesafe.akka',
            name: 'akka-actor_2.10',
            version: '2.4-SNAPSHOT'
    compile group: 'org.scala-lang',
            name: 'scala-library',
            version: '2.10.4'
}
```

除非你已经使用了 sbt，否则这就是最简单的处理方式。Gradle 是基于 Groovy 语言的，而 sbt 是基于 Scala 语言的。使用 Gradle 编写 Scala 程序需要付出一点额

外的代价，这必然会牵扯到 Scala 编译器（使用 sbt 时是不会有这些麻烦的）。虽然有点麻烦，但这也只是添加几行代码的事情。

使用Scala语言编写程序

如果你在工作中使用 Java 语言并且还没有学习 Scala 语言，那么就应该学习这门编程语言。与所有 JVM 语言（Java 除外）相比，Scala 语言拥有强大的吸收能力。通过将面向对象编程模式和函数编程模式自然而然地结合起来，Scala 语言提供了速度更快、品质更佳的编程工具。

自 20 世纪 80 年代末起，我就使用 C 和 C++ 编写程序，而且还短期使用过 Smalltalk 语言。我很喜欢 Smalltalk 语言，并努力地想要通过某种方式使 C++ 变得更像 Smalltalk。有些人也像我一样做出这种努力。无论如何，在那个时代使用 C++，对提升编程技巧所起到的帮助微乎其微。² 即使通过 C 语言使用面向对象编程方式，也会比使用 C++ 更快。据我所知，当时没有能够替代 Smalltalk 的编程语言。使用 Smalltalk 语言快速开发了几个应用程序和工具后，我确定了“Smalltalk 程序比 C++ 程序快 20 至 25 倍”这一观点的正确性。这虽然不代表 Smalltalk 程序的效率比 C++ 程序的效率高 20 至 25 倍，但我发现 Scala 语言对于 Java 的速度优势，就像 Smalltalk 语言对于 C++ 的速度优势。一行 Scala 代码可以代替数十行 Java 代码（我指的是 24 至 36 行）是不争的事实。

我刻意地避免面面俱到地介绍 Scala 编程语言。此处展示的仅是 Scala 语言的基础知识，从而使你能够阅读本书中的代码，甚至能够自己使用 Scala 语言编写入门级程序。如果你已经熟悉了面向对象程序设计语言（如 Java 或 C#）并学过一点函数式编程语言或使用过函数式编程语言，就会发现 Scala 语言非常易于掌握。

如果你想快速学会 Scala 语言，还可以参阅 Bruce Eckel 和 Dianne Marsh 撰写的 *Atomic Scala* [Atomic-Scala]。许多人（包括 Scala 语言的发明者 Martin Odersky 教授）认为，Cay S. Horstmann 撰写的 *Scala for the Impatient* 也是一本介绍 Scala 语言的好书 [Horstmann]。据我估计，大多数有经验的程序员只要花一两个周末阅读这本书，就能够在工作中使用 Scala 语言。大家的学习方式不尽相同，而且目标也各不相同。Joshua D. Suereth 撰写的 *Scala in Depth* 也是值得阅读的介绍 Scala 语言的书籍之一 [Suereth]。你还可以参加 Coursera [Coursera] 提供的网上课程，和 Typesafe 公司 [Typesafe] 及其培训与咨询合作伙伴举办的课程。

² 我并非有意贬低 C++，我的许多朋友和同事都使用这门语言并认为它是一门极为优秀的编程语言。

注意，我建议你不要接触复杂的软件库（如 `scalaz`），至少在刚开始学习 Scala 语言时不要这样做。你不应仅为了符合某种编程模式，而使用软件库。因此，如果你还不能掌握 `Functor` 和 `Monad` 软件库，就先不要接触 `scalaz` 库。通过学习本章和本书的内容，你会发现通过 Scala 语言自带的功能也可以提高工作效率。掌握了 Scala 语言的基础功能后，接触 `scalaz` 库及其提供的纯函数式数据结构和和其他高级库与语言功能才有意义。

Scala 概要教程

你是否已经做好了阅读 Scala 程序和编写 Scala 代码的必要准备？本书仅会介绍 Scala 语言的要点，而不会面面俱到地介绍其中的细节，因为这会浪费 Scala 初学者的时间。注意，本书主要介绍基于对象的开发模式，并将 Scala 语言与 Java 和 C# 做比较。尽管可以将 Scala 源代码用作脚本，但是我不会使用 Scala 语言处理脚本。本书仅是面向 Java 或 C# 开发者的 Scala 入门教程。要详细了解 Scala 语言，可参阅前面推荐的书目。

我还有意在本章避免介绍 Actor 模型（至少在大多数情况下）。如果你已经掌握了 Scala 语言的基础知识，就可以阅读下面介绍 Akka 框架的内容了。

Scala 源代码文件的扩展名为 `.scala`。下面的源代码文件含有 `Claim Check` 模式的样本：

```
ClaimCheck.scala
```

所有 Scala 类（`class`）和对象（`object`）³ 都必须放置在软件包中，而软件包必须在 Scala 源代码文件的开头声明和命名。下面的代码声明了 `Claim Check` 样本的源代码文件位于 `claimcheck` 软件包中：

```
package co.vaughnvernon.reactiveenterprise.claimcheck
```

注意，不必像使用 Java 那样在软件包语句的末尾添加分号。一般而言，在许多 Scala 代码中不必使用分号。这会使你有如去掉了枷锁一样感到轻松！如果你是一位 C# 开发者，就可以使用软件包实现命名空间的功能，这会使你有宾至如归的感觉。

```
package co.vaughnvernon.reactiveenterprise.claimcheck {  
  ...  
}
```

³ 稍后会介绍 Scala 语言的对象类型。现在你可以将它们视为指定类的单例伴生对象。

如果你想使用其他源代码文件或软件包中定义的类、对象和其他 Scala 数据类型，就必须使用 import 语句引用它们。Scala 语言的 import 语句与 Java 的 import 语句非常相似。

```
import akka.actor.Actor
import akka.actor.Props
```

// 或者

```
import akka.actor._
```

实际上，除了不必在语句末尾添加分号外，引用某个类（如 akka.actor.Actor）的方式与使用 Java 的 import 语句的方式相同。第二个 import 语句示例导入了 akka.actor 软件包中的所有类、对象等（其中包括 Actor 和 Props 类与对象）。在 Scala 语言的 import 语句中，_ 的作用与 Java 的 import 语句中的 .* 的作用相同。

应该在 Scala 源代码文件中添加什么内容呢？首先，就像 Java 和 C# 支持类一样，Scala 语言也支持类并对该功能做了一点提高。

```
class ItemChecker {
    ...
}

abstract class ItemContainer {
    ...
}

class ShoppingCart extends ItemContainer {
    ...
}
```

第一个定义的类 (ItemChecker) 不会扩展任何特定的基类。另一方面，ShoppingCart 扩展了抽象基类 ItemContainer。此处没有介绍这些类的细节。ItemChecker 和 ShoppingCart 类都具有的特点之一是，都拥有无参数构造器。这意味着可以使用下面的方式实例化 ShoppingCart 类：

```
val shoppingCart = new ShoppingCart()
```

ShoppingCart 实例的引用存储在变量 shoppingCart 中，shoppingCart 被声明为没有指定具体类型的 val 变量。这就引出了几个 Scala 语言的概念。首先，实例的引用既可以存储在 val 变量中也可以存储在 var 变量中。一旦 val 变量

被赋予了实例的引用，该 `val` 变量就无法被再次赋值。

```
val shoppingCart = new ShoppingCart
```

```
val differentCart = new ShoppingCart
```

```
shoppingCart = differentCart // 该表达式是无效的；会导致编译错误
```

另一方面，如果 `shoppingCart` 是一个 `var` 变量，那么完全可以对其进行反复赋值。

```
var shoppingCart = new ShoppingCart
```

```
val differentCart = new ShoppingCart
```

```
shoppingCart = differentCart // 该表达式有效
```

注意，在这两个 `shoppingCart` 变量声明语句中，并没有确切地指定数据类型，仅指定了引用。这是因为可以通过 `Scala` 语言的类型推断功能解决这些麻烦。类型推断是指 `Scala` 编译器能够分析代码，并检测出隐含的数据类型。该功能得出的结果与以明确方式声明类型的作用相同。

```
val shoppingCart: ShoppingCart = new ShoppingCart
```

下面介绍构造器。这里展示了为 `ShoppingCart` 类添加构造器参数的方式（使用 `Scala` 语言调用类参数的方式）：

```
class ShoppingCart(val maximumItems:Int) extends ItemContainer {  
  ...  
}
```

现在当 `ShoppingCart` 对象的客户端创建实例时，就必须向 `ShoppingCart` 对象发送 `Int` 类型的参数。

```
val shoppingCart = new ShoppingCart(50)
```

上面的构造器示例展示了 `Scala` 语言的高效功能之一。`maximumItems` 不仅作为必要的构造器参数起作用，它还在该实例的外部变得可读。也就是说，所有 `ShoppingCart` 实例都能够读取 `Int` 类型的 `maximumItems` 参数。

```
val maximumShoppingCartItems = shoppingCart.maximumItems
```

此外，因为 `maximumItems` 是一个 `val` 变量，所以无法对其再执行赋值操作。

因此，当将 `maximumItems` 声明为 `val` 变量时，`maximumItems` 变量的值就只能被读取，而不能被客户端修改或者在该类的内部被修改。

```
class ShoppingCart(val maximumItems: Int) extends ItemContainer {
  ...
  maximumItems = 100 // 该表达式是非法的；会导致编译错误
}
```

实际上，如果想要使用可变的 `maximumItems` 参数声明 `ShoppingCart` 类，可使用 `var` 代替 `val`。

```
class ShoppingCart(var maximumItems: Int) extends ItemContainer {
  ...
  maximumItems = if (maximumItems < 10) 10 else maximumItems
}
```

在这个例子中，`var` 声明语句使得 `maximumItems` 变量（代表商品数量上限）在该类的内部和外部既可以被读取也可以被修改。这段代码确保了每位顾客至少可以购买 10 件商品。然而，这样使用 `var` 变量可能不符合实际情况。`ShoppingCart` 类（代表购物车）可能一直都会使用 `val` 类型的 `maximumItems` 参数，永远不会允许该类外部的代码修改该值。实际上，你会发现需要将构造器参数设置为 `var` 变量的情况极少。

Scala 语言的类（如 `ShoppingCart`）还有一个奇妙的特点。当创建 `ShoppingCart` 类的新实例时，方法外部的所有表达式都会被执行。类内部的表达式（不论是使用 `val` 或 `var` 引用初始化的，还是用于调用新初始化的对象或其他对象中方法的），都是构造过程的组成部分。例如，当 `maximumItems` 字段被设置了初始值后，下面的表达式就会被执行：

```
...
maximumItems = if (maximumItems < 10) 10 else maximumItems
```

请思考这段代码，其中的每个表达式都有计算结果。当条件语句的 `if` 部分得出的结果为 `true` 时，字段 `maximumItems` 会被赋予 10。否则，条件语句的 `else` 部分会被计算，所得到的结果会被赋予 `maximumItems` 变量。

前面介绍了两种声明构造器参数（也称为实例字段）的操作。下面是第三种声明方式。

```
class ShoppingCart(
```

```

    catalogSource: CatalogSource,
    val maximumItems: Int)
  extends ItemContainer {
    ...
  }

```

这段代码重新将 `var` 类型的 `maximumItems` 设置为 `val` 变量，但是请注意，`catalogSource` 既不是 `val` 变量也不是 `var` 变量。这意味着这个 `Scala` 程序不会为 `catalogSource` 变量生成字段和任何类型的访问器方法（包括读取和写入方法）。因此，`ShoppingCart` 对象只能暂时使用 `catalogSource` 对象。

```

class ShoppingCart(
    catalogSource: CatalogSource,
    val maximumItems: Int)
  extends ItemContainer {
    val catalog: Catalog =
      catalogSource.catalog(catalogSource.name) getOrElse
      DefaultCatalog()
    ...
  }

```

你真正想要从 `CatalogSource` 对象获取的，是它的 `Catalog` 字段。`catalogSource.catalog` 字段会返回一个 `Option` 对象，该对象的值不是 `Some` 就是 `None`。`Option` 对象是一种值封装器，用于避免返回空值。如果 `Option` 对象的值为 `Some`，那么 `getOrElse` 方法中的读取部分就会返回 `Some` 中包含的值。否则，`getOrElse` 方法中的另一部分就会返回 `DefaultCatalog` 类的实例。

`Scala` 语言还有一些奇妙的功能。例如，在某些情况中无须使用点和括号就可以调用各种方法（如 `CatalogSource` 对象的 `name()` 方法），这是 `Scala` 语言的便捷功能之一。该功能称为中缀表示法（`infix notation`）。

那么缺少了 `new` 原语的 `DefaultCatalog` 构造器会怎样呢？实际上，下面的表达式不会直接引用该构造器：

```
val catalog: Catalog = DefaultCatalog()
```

更确切地说，这个 `DefaultCatalog` 是一个伴生对象，伴生对象是 `Scala` 语言中的一个特殊概念，而且它永远都会是单例对象；换言之，程序中仅会存在一个该类型的对象。在本例中，`DefaultCatalog` 是一个类，而它的伴生对象就像一个工厂，用于获取 `DefaultCatalog` 实例。当然，伴生对象还能够完成其他任务。下面是一段简单的实现代码：

```
object DefaultCatalog {
  ...
  def apply = new DefaultCatalog(catalogConfig)

  def name = catalogConfig.name
}
```

当你使用 `DefaultCatalog()` 表达式时, `apply` 方法就会被调用, 如上面的例子所示。 `apply` 方法仅会返回一个 `DefaultCatalog` 类的新实例, 而不会要求客户端必须了解 `catalogConfig` 参数的情况。可以在伴生对象中定义任何种类的方法(如 `name`)。为了使用该方法, 客户端会通过下面的方式使用伴生对象:

```
val catalogName = DefaultCatalog.name
```

这种方式没有问题, 但使用伴生对象的动机是什么呢? `Scala` 不支持静态方法, 因此伴生对象提供了对其伴生类所需的各种工厂和实用方法的支持。

下面介绍看起来有点奇怪的方法定义。为什么每个方法定义后面都跟着 `=` 呢? 定义方法范围的标准花括号在哪里? 返回类型和语句在哪里?

每个 `Scala` 方法基本上都是一个代码块, 这是一种引用一段代码执行一个操作的方式。因此, 可以将一个代码块赋予某个已命名的引用。`=` 左侧是引用的名称, 该引用通过 `=` 被执行赋值操作, 从而保存 `=` 右侧的代码。事实上, 如果代码块仅是单个表达式, 那么就不必使用花括号将其括起来。然而, 如果代码块中含有两个或多个表达式, 那么就必须使用花括号将它们括起来。可以使用下面的方式定义上述方法:

```
object DefaultCatalog {
  ...
  def apply: Catalog = {
    new DefaultCatalog(catalogConfig)
  }
  def name: String = {
    catalogConfig.name
  }
}
```

不仅可以使使用花括号封装代码块, 还可以通过在引用后面添加 `:type`, 声明方法会返回哪种类型的数据。明确声明方法返回数据的类型是最好的 `Scala` 编程习惯。`apply` 引用的代码块会返回 `Catalog` 类型的实例, 而 `name` 引用的代码块会返回 `String` 类型的实例。

方法怎样将计算出的结果返回给它的调用者呢？实际上，你不必了解也不会用到向调用者返回结果的语句，⁴Scala 语言功能会自动寻找方法的返回结果（将最后一个表达式的结果视为方法的返回值）。因此，`apply()` 方法的结果就是新建的 `DefaultCatalog` 对象，使用 `catalogConfig` 对象的 `name()` 访问器获得的结果就是调用 `name()` 方法取得的结果。这个功能非常好，能够通过更少的代码获得更好的效果。

这样就可以在 `ShoppingCart` 类中定义任意数量的适用于电子购物车的方法。

```
class ShoppingCart(  
  catalogSource: CatalogSource,  
  val maximumItems: Int)  
extends ItemContainer {  
  
  import scala.collection.immutable.Vector  
  
  val catalog: Catalog =  
    catalogSource.catalog(catalogSource.name) getOrElse  
    DefaultCatalog()  
  
  private var itemsContainer = Vector.empty[Item]  
  
  ...  
  def add(item: Item): Unit = {  
    itemsContainer = itemsContainer :+ item  
  }  
}
```

应注意，类主体中的所有表达式都是在构建过程中被执行的。因此，`itemsContainer` 变量会被初始化为一个空的 `Vector` 集合，用于存储 `Item` 实例。然而，为什么要将 `itemsContainer` 声明为私有变量呢？实际上，仅从类范围方面讲，导入的 `Vector` 集合是不可变的。因为其引用本身必须是可变的，所以为了使外部代码不能直接访问该变量，需要将该变量声明为私有变量。那么 `Vector` 集合的引用为什么必须是可变的呢？从 `add(Item)` 方法的实现代码中可以找到该问题的答案。要将新选择的 `Item` 对象附加到 `itemContainer` 变量中，就必须允许 `Vector` 集合生成新的 `Vector` 对象，以便存储增加了新 `Item` 对象的内容。`itemContainer` 引用必须能够被改变，以便存储指向新生成的 `Vector` 集合的引用。

⁴ 实际上，如果你在 Scala 程序中使用了向调用者返回结果的语句，会引发莫名其妙的问题，因为这会改变你编写程序的目标。因此，不推荐使用向调用者返回结果的语句。要了解详情，请参阅 <http://tpolecat.github.io/2014/05/09/return.html>。

这个技巧很有趣，但是会不会因为要添加新的 `Item` 对象而总是要创建新的 `Vector` 集合，因而大幅度增加系统开销呢？毕竟可变集合的处理难度较高。不必担心这一点。实际上，在添加新 `Item` 对象的过程中不会创建全新的 `Vector` 对象。作为一门支持函数编程模式（其中的元素都是不可变的）的语言，Scala 中的不可变集合⁵已经通过优化去除了系统开销。在添加新 `Item` 对象时，实际上新的 `Item` 集合元素会被添加到已存在的数据结构中，但是这只会改变当前的 `Vector` 对象而不会改变原 `Vector` 对象的引用，因此拥有原 `Vector` 对象引用的对象无法看到新添加的 `Item` 对象。只有拥有 `Vector` 集合引用的新版本的对象才能看到新添加的 `Item` 对象。

请注意 `add(Item)` 方法。该方法的返回类型被定义为 `Unit`，这是在 Scala 中表明返回值没有意义的一种方式。该类型就像 Java 和 C# 中的 `void` 类型。返回 `Unit` 类型值的方法不能在表达式链条中被使用。

还应注意 `Vector` 集合类范围中的导入语句。在 Java 中，总是会将导入语句添加到文件的顶部，因为 `.java` 文件只能含有一个全局变量类。我想这个原因你已经猜到了。Scala 允许在一个源代码文件中包含多个全局变量类、伴生对象等。因为一个源代码文件中会有多个类、对象，所以它们会拥有本身的导入语句。导入语句可以放在由花括号封装的任何种类的代码块中，如类的主体、方法的定义等类型的代码块。

如果你使用过 Java，可能会因无法将符号用作方法的名称而感到有一点遗憾。Scala 改变了这一点，在 Scala 中使用操作符甚至比在 C++ 中更方便。当然，不应滥用这个功能。没有限制，自由也不会存在。例如，在 Scala 中可以将 `!` 用作方法的名称。

```
def !(message: Any): Unit = {
    ...
}
```

实际上，下面是一种发送 Akka 消息的方式：

```
actor ! SomeMessage()
```

使用 `!` 方法的方式与使用 `tell()` 方法的方式相同。而且，还不需要在接收者 `actor` 对象后面加圆点，也不需要使⽤括号将参数 `SomeMessage()` 括起来，这使代码更易于阅读得多。

⁵ Scala中也有可变集合。可变集合和不可变集合分别位于两个不同的软件包中。

实际上,下面不是!方法的全部定义,然而却是一个该方法的完整定义:

```
def tell(message: Any, sender: ActorRef): Unit = this.!(message)(
  sender)

...
def !(message: Any)(implicit sender: ActorRef=Actor.noSender): Unit = {
  actorCell.sendMessage(message, sender)
}
```

当你使用!或tell()方法向actor对象发送消息时,还需要指明消息的发送者。前面的代码表明发送者(sender)位于第二对括号和隐式参数定义之间。Scala编译器会在当前代码上下文中寻找implicit val some Name: ActorRef的定义。因此,可以将该变量的名称设置为sender,也可以使用任何名称设置该变量。

```
implicit val sender: ActorRef = sender()
```

该定义确保了当!或tell()方法被调用时, sender 变量会被作为参数传送。如果当前代码上下文中没有这个类,那么默认引用(即Actor.noSender)就会被使用。

如tell()方法的实现代码所示,如果你想为隐式声明的方法参数提供显式值,就必须使用第二对括号。

```
... this.!(message)(sender)
```

这可能会滥用隐式参数,因而使代码变得不易阅读。谨慎地使用隐式参数,会大幅度提高代码的可读性。

Scala 源代码文件中还可以定义特征(trait)。Scala 中的特征有点像 Java 和 C# 中的接口和抽象类。它像一种接口,会在内部定义为使用它们的类的对象定义协议。它还与抽象类相似,因为特征内部至少会有一部分拥有默认的实现代码。从这方面看,特征就是一种抽象类。如果特征被使用了,那么它就会变成抽象类。事实上,多重特征可以由单个类实现或扩展。该操作通常称为混入多重特征。

```
class ShoppingCart(
  val catalogSource: CatalogSource,
  val maximumItems: Int)
extends ItemBrowser with ItemContainer {
  ...
}
```

在这段代码中,ItemBrowser 和 ItemContainer 都是混入的特征。第一

个特征或超类是使用 `extends` 关键字混入的。所有继承的特征都是使用 `with` 关键字混入的。这两个特征都声明了抽象值。`ShoppingCart` 构造器参数用于初始化在这些特征中声明的变量。

```
trait ItemBrowser {
  val catalogSource: CatalogSource // 抽象值
  ...
}

trait ItemContainer {
  val maximumItems: Int // 抽象值
  ...
}
```

前面介绍过 `Scala` 类的种类与 `Java` 和 `C#` 中类的种类相似。除了那些相似的类外, `Scala` 还有一种样本类 (`case class`)。

```
case class ProcessOrder(orderId: String)

val message = ProcessOrder("123")
```

这是非常好的功能。在为样本类创建新实例时, 甚至不必使用关键字 `new`。但是样本类的优点还未尽于此。

我认为样本类是世界上创建不可变值对象 [IDDD] 的最简单方式。这是为什么呢? 在样本类 `ProcessOrder` 的定义中, 类参数 (构造器参数) 仅像普通的 `Scala` 类参数那样被定义。即使你没有明确地定义, `orderId` 字段也会自动被声明为 `val` 变量。此外, `Scala` 编译器能够自动生成多个方法: `equals()`、`hashCode()`、`toString()`、`copy(...)`, 还能够为被作为类参数 (如 `orderId`) 命名的所有字段自动生成公用的读取访问器。熟悉领域对象实现方式的 `Java` 和 `C#` 程序员, 都知道反复创建样板代码是怎样的痛苦。这个难题也被 `Scala` 通过代码环保功能解决了。通常, 可以使用样本类创建在 `Actor` 对象之间传递的不可变消息。因为样本类会创建完美的值对象 [IDDD], 所以除了被用作消息类型外, 它们还有许多用途。而且, 样本类还提供了一种执行模式匹配 (稍后会详细介绍) 的便捷途径。

如果你想在样本类中添加自定义行为, 其处理方式就像处理普通类一样简单。

```
case class ConfigureProcessor(
  orderProvider: OrderProvider,
  timeOut: Long) {
```

```
def timeoutAsDuration(): Duration = {
    Duration.create(timeout, TimeUnit.MILLISECONDS)
}
}
```

此刻你可能想知道 **Scala** 是怎样支持循环和迭代操作的。实际上，**Scala** 提供了非常多的循环方式。本书仅会列举几个示例。下面是编写 **for** 循环的一种方式：

```
for (counter <- 1 to 20) {
    println(counter) // 变量 counter 每次都被赋予 1 至 20 之间的新值
}
```

这个 **for** 表达式会重复执行 20 次（从 1 到 20），每次都会使用当前循环次数值设置 **var** 变量 **counter**。如果将这段代码中的 **to** 改为 **until**，那么这个 **for** 表达式被重复执行的次数就会小于 20 次，而且变量 **counter** 被赋予的值只能为 1 至 19。

```
for (counter <- 1 until 20) {
    println(counter) // 变量 counter 每次都被赋予 1 至 19 之间的新值
}
```

使用 **for** 表达式可以轻松迭代集合。

```
for (element <- Vector(1,2,3,4,5)) {
    println(element) // element 变量每次都被赋予 1 至 5 之间的新值
}
```

你可以使用集合本身提供的迭代操作，还可以提供处理每个集合元素的闭包。

```
Vector(1,2,3,4,5) map { element => println(element) }
```

下面的示例使用了 **for** 推导语句：

```
val numbers = Vector(1,2,3,4,5,6,7,8,9,10)

val evenNumbers = for (number <- numbers) {
    if (number % 2 == 0)
} yield number
```

这个执行 10 次（从 1 至 10）的迭代操作会过滤掉所有无法被 2 整除的数值。通过 **if(...)** 过滤器的数值会成为计算结果的组成部分。该计算结果（即

包含 2、4、6、8 和 10 的 Vector 集合) 会被变量 evenNumbers 引用。因为该迭代操作的源操作数是 Int 类型的 Vector 集合, 所以即使这段程序没有明确声明 evenNumbers 变量的类型, 该推导语句也知道应将计算结果赋予 Int 类型的 Vector 集合变量。

还可以通过另一种方式获得相同的结果, 即使用更为简洁的 Scala 代码:

```
val evenNumbers =
  for {
    number <- numbers
    if number % 2 == 0
  } yield number
```

如果你想完整地声明 evenNumbers 引用, 可使用下面的方式:

```
val evenNumbers: Vector[Int] = ...
```

这段代码声明了一个类型为 Int 的 Vector 集合。Java 和 C# 使用尖括号表示泛型 (即 <type>), 与 Java 和 C# 不同, Scala 使用方括号表示泛型 (即 <type>)。

模式匹配功能是 Scala 中功能最强大的工具之一, 可以在多种环境 (如由数字元素组成的集合中) 中使用它。

```
Vector(1,2,3) map {
  case 1 => println("One")           // 显式编号为 1 的单词
  case 2 => println("Two")           // 显式编号为 2 的单词
  case 3 => println("Three")         // 显式编号为 3 的单词
}
```

执行匹配操作时不必对集合进行迭代。对单个对象也可以进行模式匹配, 在下面的例子中该单个对象为 10。

```
10 match {
  case 1 => println("One")
  case 2 => println("Two")
  case 3 => println("Three")
  case _ => println("Several") // 显示默认的匹配单词
}
```

注意, 当 Actor 对象收到消息时就会用到模式匹配功能。这是 Actor 对象分辨它所收到消息是哪种类型的途径。在处理消息时, 在匹配情况中通常需要使用类型和结果参数。我强烈建议你使用样本类 (如 ConfigureProcessor 和 ProcessOrder) 定义消息类型。

```
case class ConfigureProcessor(
  orderProvider: OrderProvider,
  timeout: Long)

case class ProcessOrder(orderId: String)

val message = ProcessOrder("123")

message match {
  case config: ConfigureProcessor =>
    configureProcessor(init)
  case processOrder: ProcessOrder =>
    val order = orderFor(processOrder.orderId)
    ...
}
```

这段代码展示了将匹配结果添加到参数中的一种方式，这样就可以在与匹配情况有关的表达式中使用这些匹配结果了。在这段代码中，config 引用了消息实例 ConfigureProcessor，而 processOrder 引用了消息实例 ProcessOrder。另一种添加匹配结果的方式是命名类参数。

```
...
message match {
  case ConfigureProcessor(orderProvider, timeout) =>
    configureProcessor(orderProvider, timeout)
  case ProcessOrder(orderId) =>
    val order = orderFor(orderId)
    ...
}
```

尽管此处没有进行面面俱到的介绍，但本书提供的知识足以使你理解代码示例并自己动手编写 Scala 程序。

使用Akka框架编写程序

如前所述，使用常规方式编写多线程程序，会使问题过度复杂化，因而难以获得成功。旧的编程工具和模式，已经变成我们充分利用现代硬件发展进步优势（如不断增加的处理器和处理器核心数量，以及处理器高速缓存的容量）的障碍。这导致编写事件驱动的、可伸缩的、有韧性的响应式应用程序，通常显得过于困难和危险，因而使大家通常避免编写这类程序。

Akka 工具包应运而生，它专门用于处理普通多线程、分布式计算和容错编

程方式无法处理的问题。Akka 框架通过使用 Actor 模型做到这一点，该模型提供了功能强大的抽象，使并发和并行程序更易于推导和编写。这并不是说使用 Akka 框架就可以不用思考并发处理步骤，仍旧必须设计并行模式和延迟方式，而且还要考虑应用程序的状态并使应用程序避免出现不必要的锁定情况。Akka 框架和 Actor 模型的作用是解决常见的并发编程问题，如死锁、活锁、低效代码和不应将线程用作第一类编程模型。简言之，Akka 框架使并发软件设计变得尽可能简单，并更易于取得成功。

下面几节将介绍使用 Akka 框架和 Actor 模型开发响应式应用程序的基础知识。你会了解 ActorSystem、Actor、ActorRef 和 ActorContext 类，以及监督机制、远程处理模式、集群处理模式和测试技巧。掌握了这些知识后，你就能使用企业级响应式应用程序和整合模式了。

Actor 系统

每个 Akka 应用程序都必须创建一个名为 ActorSystem 的类。这个类代表 Actor 系统，其中包含了 Actor 对象的层次结构，该系统中的所有 Actor 对象都会使用同一套配置。下面的代码在本地 JVM 中创建了一个名为 ReactiveEnterprise 的 ActorSystem 对象：

```
import akka.actor._
...
val system = ActorSystem("ReactiveEnterprise")
```

这会使用默认配置创建已命名的 ActorSystem 对象。Actor 系统配置包括 Actor 对象设置（包括消息缓存类型和访问远程 Actor 对象的方式）、在线程中执行 Actor 对象的调度器的声明以及其他系统和 Actor 属性。Akka 工具包倾向于使在不同环境中会改变的所有应用程序值能够被配置，而不是在 Actor 对象中以硬编码方式处理这些值。这在测试过程中特别有用。

彻底了解配置

下面举一个 Akka 配置的小例子。这段代码通常应放置在 application.conf 文件中，但也可以通过多种不同方式进行定义。

```
# application.conf for ActorSystem: RiskRover
akka {
  # default logs to System.out
  loggers = ["akka.event.Logging$DefaultLogger"]
```

```
# Akka configured loggers use this loglevel.
# Use: OFF, ERROR, WARNING, INFO, DEBUG
loglevel = "DEBUG"

# Akka ActorSystem startup uses this loglevel
# until configs load; output to System.out.
# Use: OFF, ERROR, WARNING, INFO, DEBUG
stdout-loglevel = "DEBUG"

actor {
  # if remoting: akka.remote.RemoteActorRefProvider
  # if clustering: akka.cluster.ClusterActorRefProvider

  provider = "akka.actor.LocalActorRefProvider"

  default-dispatcher {
    # Default Dispatcher throughput;
    # set to 1 for as fair as possible,
    # but also poor throughput
    throughput = 1
  }
}
```

这段代码的内容丰富程度足以使你完成单个 Actor 对象和整个 Actor 系统的配置，而且其中的某些部分也超出了本书涵盖的范围。实际上，其内容已经超出了 Akka 文档的范围，其中还包含了 Typesafe Config Library 文档的内容。要完全掌握这些内容，你既需要参阅 Akka 文档，也需要参阅 Typesafe Config Library 文档。除非另有说明，否则这就是本书所有示例程序的默认配置。

你可以在一个应用程序中创建多个 ActorSystem 实例，但一般来说，每个应用程序中只需创建一个 ActorSystem 对象。⁶ 如果你想使一个 ActorSystem 对象与另一个 ActorSystem 对象协作，可使用 Akka 框架的远程功能；请参阅本章稍后介绍的有关远程处理模式的内容。如果你使用了 Akka 框架的集群功能，就可以将一个 ActorSystem 对象扩展到多台 JVM 中；请参阅本章稍后介绍的有关集群处理模式的内容。

当 Actor 系统被创建时，有几个 Actor 对象会随着它一起被创建。其中包括 *root* 守护对象、*user* 守护对象和 *system* 守护对象。这些对象是 Actor 系统中所有

⁶ Play 框架会创建本身的 ActorSystem 对象，因此如果你使用了 Play 框架，你编写的应用程序中就会总是有两个 ActorSystem 实例。一般来说，不应使用 Play 框架提供的那个 ActorSystem 实例。

其他 Actor 对象的最高监督者，而应用程序创建的 Actor 对象位于 user 守护对象的下方，如图 2.1 所示。

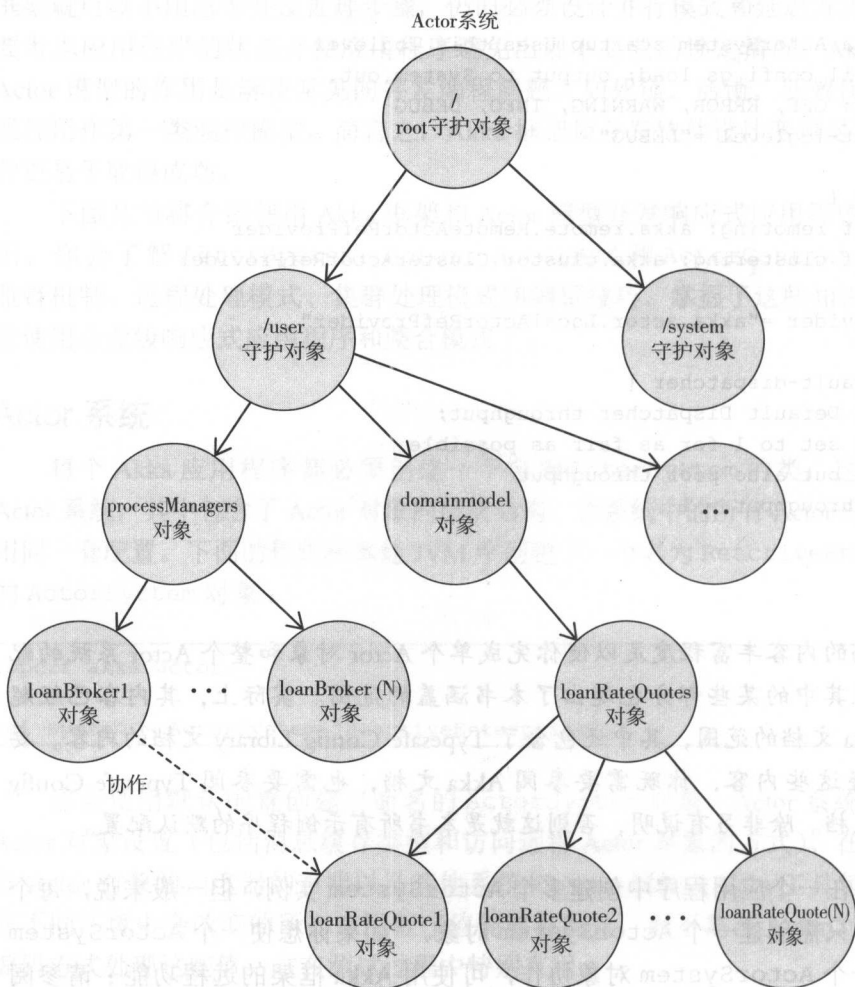


图 2.1 Actor 系统中含有 root 守护对象、user 守护对象和 system 守护对象。应用程序创建的所有 Actor 对象都位于 user 守护对象的下方。

第 1 章着重介绍过，监督机制是 Actor 模型的原始特性之一。监督机制是 Akka 框架提供的重要扩展，它能够帮助你开发出具有高度韧性的应用程序。基本上，任何创建了子 Actor 对象的 Actor 对象，都会自动变为其子 Actor 对象的监督者。如果子 Actor 对象崩溃了（例如抛出了异常），那么它的父 Actor 对象就必须在执行下列哪种操作之间做出选择：使子对象继续运行、重启子对象、停止子对象或使失效情况升级。前三种操作是父对象执行的恢复操作，而使失效情况升

级的操作会将控制权转交给父对象的父对象（即崩溃子对象的祖父对象）。稍后会详细介绍监督机制。

要直接在 `user` 守护对象下方创建 `Actor` 对象，可通过 `system.actorOf()` 方法使用 `ActorSystem` 对象，如下面的例子所示：

```
import akka.actor._
...
val system = ActorSystem("ReactiveEnterprise")
...
// 创建 Actor 对象并获取该对象的 ActorRef 引用
val processManagersRef: ActorRef =
    system.actorOf(Props[ProcessManagers], "processManagers")

// 使用 ActorRef 引用向 Actor 对象发送消息
processManagersRef ! BrokerForLoan(banks)
```

为什么要使用！命名方法？

如前所述，Scala 语言允许将符号用作方法的名称。除了将符号用作方法的名称外，`!` 方法基本上与 `tell()` 方法相同。我特别喜欢使用 `!` 命名方法，因为这种方式更为直观并且比 `tell()` 方法使用的代码更少。

还应注意，如果你使用了 `tell()` 方法，就需要传送两个参数，即消息实例和发送者的 `ActorRef` 引用。

```
processManagersRef.tell(BrokerForLoan(banks), self)
```

然而，在使用 `!` 方法时，发送者的 `ActorRef` 引用会成为一个隐式参数，这意味着你不必亲自发送它。Scala 编译器会自动寻找代码上下文中的隐式 `ActorRef` 引用，找到该参数后，会自动将该值用作传送参数。本书稍后会介绍隐式参数。

`ActorSystem` 对象的 `actorOf()` 方法会创建一个名为 `processManagers`、类型为 `ProcessManagers` 的 `Actor` 对象。当你命令 `ActorSystem` 实例创建新的 `Actor` 对象时，`ActorSystem` 实例不会返回 `Actor` 实例，而会返回 `ActorRef` 引用（在本例中，`ActorRef` 引用被存储在 `processManagersRef` 变量中）。这种隔离机制在真正的 `Actor` 对象和 `Actor` 对象的客户端之间创建了一个间接层。当客户端使用 `processManagersRef` 变量向名为 `processManagers` 的 `Actor` 对象发送消息（如 `BrokerForLoan` 消息）时，`ActorRef` 引用的作用仅是将入队的消息添加到 `Actor` 对象的消息缓存中。

保存 ActorRef 引用的变量的命名惯例

将保存 ActorRef 引用的变量命名为 `processManagersRef` 并不是硬性规定。实际上, 将之命名为 `processManagers` 会更好。将代表 ActorRef 的 Ref 用作保存 ActorRef 引用的变量的名称的后缀, 是为了凸显该变量代表的含义。这样做主要是为了避免混淆保存 Actor 对象的变量和保存 ActorRef 引用的变量。

为保存 ActorRef 引用的变量的名称添加 Ref, 可以使该变量名称变得更易于阅读并更能体现它代表的抽象 (`processManagers` 对象) 的含义。我会一直使用这个命名惯例。

`processManagers` 对象的父对象是名为 `user` 的 Actor 对象, 它是一个守护对象。`user` 守护对象是一个自动防故障组件, 用于防止当直接子对象 (如 `processManagers`) 崩溃时, Actor 系统出现灾难性故障。

因此你可以随时为 `user` 对象创建子 Actor 对象。但是请想象一下, 当你直接在 `user` 对象下面放置了数千甚至数百万个 Actor 对象, 会形成怎样的情况。实际上此处的“放置”一词对这种情况描绘得并不全面, 但该词也说明了大部分事实。数量庞大的 Actor 对象会使系统的某些部分处于停滞状态。因此, 最好为 Actor 对象划分出等级。但这不是创建 Actor 对象层次结构的唯一原因。这样做还可以防止 `user` 守护对象直接面对应用系统的失效情况。因此, 应该将 `processManagers` 用作 Actor 系统中顶级的 Actor 对象和最高级的监督者。`processManagers` 对象的作用仅是创建和监督指定类型的处理过程管理器。而它本身并不起处理过程管理器的作用。这进一步证实不应直接在 `user` 守护对象下方创建许多 Actor 对象。最好为指定类型的 Actor 对象创建一个监督者对象, 并为粗粒度处理过程、任务或领域模型实体单独创建监督者。

如图 2.1 所示, 可以将 `processManagers` 和 `domainModel` 都创建为最高级监督者。Actor 对象 `domainModel` 被用作代表领域模型的 Actor 对象的父对象和监督者 [IDDD]。例如, `domainModel` 对象的下方是 `loanRateQuotes` 对象 (是由 `domainModel` 创建的), `loanRateQuotes` 对象 (在银行业务模型中代表贷款利率报价) 是所有 `loanRateQuote` (1 至 N) 对象 (在银行业务模型中代表单项贷款利率报价) 的监督者。每个 `LoanRateQuote` 对象都与一个 `LoanBroker` 对象 (在银行业务模型中代表贷款经理) 对应, 这使 `LoanBroker` (1 至 N) 对象 (在银行业务模型中代表贷款代理人) 能够通过 `loanRateQuote` (1 至 N) 对象管理每家银行的贷款利率报价。所有这些层次结构会帮助你使应用程序中的 Actor 对象有条理、有韧性和使整个系统获得最佳性能。

为 Actor 系统创建了层次结构后, 怎样才能找到某个 Actor 对象的引用呢?

可使用 `ActorSystem` 对象中的 `actorSelection()` 方法，从 `user` 守护对象开始向下搜索。

```
val system = ActorSystem("ReactiveEnterprise")
...
val selection = system.actorSelection("/user/processManagers")

selection ! BrokerForLoans(banks)
```

最少执行一次的传送机制

只要某个 `Actor` 对象存在，那么使用 `ActorRef` 或 `ActorSelection` 方法都可以向该对象发送消息。但是该 `Actor` 对象有可能处于正在停止或已经停止的状态中，在这些情况中消息就无法被送达。因此，如果确保能够发送指定的消息，就必须采用重新加载和重试策略。`Akka Persistence` 可以完成许多这类任务，它是一种可选的 `Akka` 工具包。第 5 章、第 7 章、第 9 章和第 10 章会介绍到这些工具，可以使用它们支持最少执行一次的消息传送机制。

`actorSelection()` 方法会返回 `ActorSelection` 选择路径，而不会返回 `ActorRef` 引用。使用 `ActorSelection` 对象可以向该路径指向的 `Actor` 对象发送消息。然而，请注意，与使用 `ActorRef` 引用的方式相比，通过这种方式发送消息的速度较慢并且会占用更多资源。但是，`actorSelection()` 方法仍旧是一个优秀的工具，因为它可以执行查询由通配符代表的多个 `Actor` 对象的操作，从而使你能够向 `ActorSelection` 选择路径指向的任意个 `Actor` 对象广播消息。

```
val system = ActorSystem("ReactiveEnterprise")
...
val selection = system.actorSelection("/user/*")

selection ! FlushAll()
```

上面的代码向所有顶级 `Actor` 对象（即 `user` 守护对象所有的直接子对象）发送了 `FlushAll` 消息。表 2.1 展示了常用的 `ActorSystem` 对象中的方法。该表没有列出 `ActorSystem` 对象中的所有方法，要了解其他 `ActorSystem` 对象中的方法，请参阅 `Akka` 文档。

表 2.1 `ActorSystem` 对象中的方法

方法	描述
<code>actorOf(...)</code>	使用该方法可以通过指定的名称（不能为空或 <code>null</code> ，也不能以 <code>\$</code> 开头）为当前上下文创建子对象。注意，不应在 <code>Actor</code> 系统的 <code>user</code> 守护对象下方直接创建过多的 <code>Actor</code> 对象

续表

方法	描述
<code>actorSelection(...)</code>	使用该方法可以执行搜索 Actor 对象的操作，从而获得路径参数 (ActorSelection 对象，其中可能会包含通配符)。因为带通配符的路径会对应多个 Actor 对象，所以 ActorSelection 对象会指向多个 Actor 对象
<code>awaitTermination(...)</code>	调用 <code>shutdown()</code> 方法后，使用 <code>awaitTermination(...)</code> 方法可以等待系统彻底结束运行
<code>deadLetters</code>	使用该方法可以获得死信通道 (Dead Letter Channel) 的 ActorRef 引用 (请参阅第 5 章)
<code>eventStream</code>	使用该方法可以获得系统的主 EventBus 特征，请参阅第 5 章
<code>isTerminated</code>	如果系统在执行了 <code>shutdown()</code> 方法后彻底停止运行了，那么调用该方法就会返回 <code>true</code>
<code>log</code>	使用该方法可以获得系统的 LoggingAdapter 对象，默认情况下会将该对象写入 eventStream 对象中
<code>name</code>	使用该方法可以获得 String 类型的系统名称
<code>scheduler</code>	使用该方法可以获得系统中用于创建计时器事件的调度器
<code>shutdown()</code>	使用该方法可以停止 user 守护对象，从而停止 user 对象的所有子对象，然后停止 system 守护对象和所有系统级的 Actor 对象，并最终停止整个系统
<code>stop(actor: ActorRef)</code>	使用该方法可以通过异步方式停止 ActorRef 引用指向的 Actor 对象，并向 user 守护对象发送一条消息，尽管该方法通过异步方式执行操作，但该方法会在内部为回复消息执行阻塞操作

现在我们对 ActorSystem 对象有了充分的了解，并且学会了与该对象进行交互的方式，下面让我们学习实现和使用 Actor 对象的方式。

实现 Actor 对象

Akka 框架中的所有 Actor 对象都必须扩展 `akka.actor.Actor` 特征。你编写的 Actor 对象至少要支持 `receive` 代码块。

```
import akka.actor._
```

```
class ShoppingCart extends Actor {
```

```
  def receive = {
```

```
    case _ =>
  }
}
```

上面的 Actor 类 ShoppingCart 没有执行什么重要的操作。该类的实例仅会接收消息并立刻回复。case _ => 表达式是一个用于处理所有类型消息的模式匹配器。因为该表达式中 => 部分后面没有代码，所以 receive 代码块中的 case 语句不会起作用，也不会返回结果。

但是，这些代码的背后另有乾坤。当 Actor 对象被启动、停止和重启时，Akka 框架会自动调用 Actor 对象生命周期中默认存在的 4 个方法。这 4 个方法是在 Actor 基类中定义的，你可以在具体 Actor 对象中重写它们。表 2.2 展示了这 4 个方法。

表 2.2 Actor 对象生命周期中默认存在的 4 个方法（按逻辑顺序排序）

方法	描述
preStart()	Actor 对象被创建后，当 Actor 对象启动时，该方法会被调用。Actor 对象是通过异步方式创建的，当 Actor 对象完全被创建成功后，Actor 对象就会启动。Actor 对象默认的实现代码是空的，不会执行任何操作。在大多数情况中，需要重写该方法，以便使 Actor 对象能够执行所有启动时执行的初始化操作
postStop()	当 Actor 对象停止运行时，该方法会被调用。当 ActorSystem 或 ActorContext 对象（Actor 对象中会含有一个 ActorContext 对象，可通过 context 方法获得该对象）中的 stop(ActorRef) 方法被调用时，Actor 对象会以异步方式停止运行。这使 Actor 对象能够清理没有被 preRestart() 方法处理的无子对象资源。大多数情况下，你应该重写该方法，以便使 Actor 对象能够在停止运行后执行清理操作
preRestart(...)	通过失效监督策略可以重启失效的子 Actor 对象。在执行重启操作的过程中，可以在重启 Actor 对象前调用该方法。该方法默认的实现代码会处理 Actor 对象停止运行前使用的资源，处置 Actor 对象的所有子对象。执行完清理操作后，postStop() 方法会自动被调用。通常不必重写这个方法。如果重写了这个方法，就必须多加小心
postRestart(...)	Actor 对象重启后会调用这个方法，使 Actor 对象能够在失效并重启后被重新初始化。该方法默认的执行代码会调用 preStart() 方法。通常不必重写这个方法。如果重写了这个方法，就必须多加小心

下面是这 4 个方法的示例代码：

```
import akka.actor._

class ShoppingCart extends Actor {
  override def postRestart(reason: Throwable): Unit {
    ...
  }

  override def postStop(): Unit {
    ...
  }

  override def preRestart(
    reason: Throwable,
    message: Option[Any]): Unit {
    ...
  }

  override def preStart(): Unit {
    ...
  }

  def receive = {
    case _ =>
  }
}
```

每个 Actor 对象都拥有一个 ActorContext 对象，通过 Actor 对象中的 context() 方法可以获得该对象。ActorContext 对象为它的所有者对象提供了一种处理方式，使它的所有者能够以安全的、非破坏性的方式使用它的所有者包含的部分基础实现代码。这个例子使用 ActorContext 对象创建子 Actor 对象 Task：

```
import akka.actor._

class TaskManager extends Actor {
  ...
  def nextTaskName(): String = {
    "task-" + ...
  }
  def receive = {
    case RunTask(definition) =>
      val task = context.actorOf(Props[Task], nextTaskName)
      task ! Run(definition)
      ...
  }
}
```

```

case TaskCompleted =>
    ...
}
}

```

通过 ActorSystem 对象创建 Actor 对象与通过 ActorContext 对象创建 Actor 对象是有区别的。如果 TaskManager 对象（代表任务管理器）的子对象使用下面的路径和名称，就说明代表单个任务的 Actor 对象是在它们的父对象和监督者下方创建的：

```
"/user/taskManagers/taskManager1"
```

如果 taskManager1 对象创建了 3 个代表子任务的子对象，那么这些子对象应拥有下面的路径和名称：

```

"/user/taskManagers/taskManager1/task1"
"/user/taskManagers/taskManager1/task2"
"/user/taskManagers/taskManager1/task3"

```

将 Actor 模型与对象模型区分开的一个因素是，Actor 对象能够以动态方式更改本身的行为。状态设计模式 [GoF] 体现了这个特点，通过基于对象的解决方案可以实现状态设计模式。然而，与基于对象的状态实现代码相比，使用 Scala 语言和 Akka 框架更容易实现这种设计模式。通过使用 ActorContext 对象，Actor 对象能够通过接收代码块从一种行为切换到另一种行为。

```

import akka.actor._

class TaskManager extends Actor {
    var statusWatcher: ActorRef = None
    ...
    override def preStart(): Unit {
        context.become(houseKeeper)
    }

    def houseKeeper: Receive = {
        case StartTaskManagement(externalStatusWatcher) =>
            statusWatcher = Some(externalStatusWatcher)
            context.become(taskDistributor)
        case StartTaskManagementWithoutStatus =>
            context.become(taskDistributor)
    }

    def taskDistributor: Receive = {

```

```

case RunTask(definition) =>
    val task = context.actorOf(Props[Task], nextTaskName)
    task ! Run(definition, statusWatcher)
    ...
case TaskCompleted =>
    ...
}
}

```

在这段代码中, TaskManager 对象既拥有管家的行为 (houseKeeper 方法), 也拥有任务分配者的行为 (taskDistributor 方法)。接收类型的偏函数拥有两种定义。TaskManager 对象通过使用 ActorContext 对象中的 become() 方法, 可以在这两种行为之间切换。在 TaskManager 对象启动前, 该对象中的 preStart() 方法会被调用。当出现这种情况时, TaskManager 对象会切换到管家的行为。这意味着此时 TaskManager 对象只能接收 StartTaskManagement 和 StartTaskManagementWithoutStatus 消息 (分别代表开始管理任务和在不状态的情况下开始管理任务), 并只能对这两类消息做出回应。处理了这两类消息中的一个后, TaskManager 对象会切换到任务分配者的行为。在存在上一个行为的情况下, 使用 ActorContext 对象中的 unbecome() 方法可以使用上一个行为替换 Actor 对象的当前行为。

使用 ActorContext 对象还可以做其他事情, 表 2.3 展示了比较重要的几项 (要详细了解这方面内容, 请参阅 Akka 文档)。

表 2.3 ActorContext 对象提供的函数

函数	描述
actorOf(...)	使用该函数可以创建子 Actor 对象。你编写的 Actor 系统中的大多数 Actor 对象, 都应该在应用程序级 Actor 对象的 ActorContext 对象的下方创建
actorSelection(...)	使用该函数可以获得 ActorSelection 选择路径, 其中可能包含通配符。因为带有通配符的路径能够代表多个 Actor 对象, 因此 ActorSelection 选择路径能够指向多个 Actor 对象
become(...)	使用该函数可以通过指定偏函数设置 Actor 对象的当前行为。该函数有两个参数, 它的第二个参数是 discardOld, 该参数的默认值为 true。当该参数的值为 true 时, 在设置当前行为前, Actor 对象的上一个行为会被丢弃。当该参数的值为 false 时, Actor 对象的上一个行为仍旧会被保存在堆栈中, 当前设置的 Actor 对象行为也会被推入堆栈, 并位于上一个行为条目的上方

续表

函数	描述
children	使用该函数可以获得 Actor 对象所有直接子对象的 ActorRef 引用的 Iterable 特征
parent	使用该函数可以获得 Actor 对象父对象的 ActorRef 引用
props	使用该函数可以获得用于创建 Actor 对象的 props 配置对象
self	使用该函数可以获得 Actor 对象的 ActorRef 引用
sender()	使用该函数可以获得消息发送者的 ActorRef 引用
stop(actor: ActorRef)	使用该函数可以停止参数 ActorRef 指向的 Actor 对象。尽管这是一种异步操作，但如果被停止的 Actor 对象是当前 ActorContext 上下文的子对象，那么该 Actor 对象的名称会立刻被释放并能够被重用
system	使用该函数可以获得 Actor 对象所属的 ActorSystem 系统
unbecome	在上一个行为存在的情况下，使用该函数可以使 Actor 对象从当前行为切换到上一个行为

应谨慎使用 sender() 函数

在使用 sender() 函数时请多加小心。这是一个函数，而不是一个 val 实例，使用它可以获得当前消息的发送者，该发送者与收到的消息处于同一个上下文中。如果你关闭了该函数，很可能会遇到难以进行调试的问题。

```
case calculateRisk: CalculateRisk =>
  context.system.scheduleOnce(1 minute) {
    sender ! UnfinishedCalculation()
  }
  ...
```

因为 sender() 是一个函数，所以在闭包求值前它不会被求值，在本例中这个等待时间为 1 分钟。如果由于使用了调度器的原因，Actor 对象已经接收了其他消息（发生这种情况的可能性很大），那么经过延迟时间后执行的 sender() 函数，可能会返回错误的 ActorRef 引用。怎样解决这个问题呢？下面是解决方案：

```
case calculateRisk: CalculateRisk =>
  val requestSender = sender
  context.system.scheduleOnce(1 minute) {
    requestSender ! UnfinishedCalculation()
  }
  ...
```


现在 `requestSender` 是一个 `val` 变量了，其值是恒定不变的，而且它永远都会保存你想要获得的 `ActorRef` 引用（在上下文中是 `CalculateRisk` 消息发送者的引用）。

监督

在介绍 Akka 框架的 `ActorSystem` 类时，我已经介绍过许多 Actor 监督机制的内容，只是还没有介绍如何真正实现监督机制。要实现监督机制，必须先了解谁直接对监督机制负责。如前所述，父 Actor 对象负责监督它的子对象。如图 2.2 所示，哪个 Actor 对象负责处理崩溃的 Actor 对象 `loanRateQuote1` 和 `loanRateQuote2` 呢？`loanBroker1` 对象通过向 `loanRateQuote1` 对象发送消息并接受回复的消息，与 `loanRateQuote1` 对象协作。这是否意味着 `loanBroker1` 对象负责监督 `loanRateQuote1` 对象呢？答案是否定的。监督 `loanRateQuote1` 对象的是它的父对象 `loanRateQuotes`，`loanRateQuotes` 对象负责监督所有代表单项贷款利率报价的对象，即 `loanRateQuote1`、`loanRateQuote2`……`loanRateQuoteN`。

当 `loanRateQuote1` 对象崩溃后，`loanRateQuotes` 对象会做什么呢？`loanRateQuotes` 对象的行为由 `loanRateQuote1` 对象崩溃的方式和已定义的监督回复策略决定。默认情况下，所有父 Actor 对象都会继承默认的 `supervisorStrategy` 属性，该属性具有下列策略。

- `ActorInitializationException`：失效的子对象会被停止。
- `ActorKilledException`：子对象会被停止。
- `Exception`：子对象会被重启。
- 任何其他类型的 `Throwable` 异常都会导致父对象将失效情况升级。

如果默认的监督策略不适合你编写的贷款利率报价应用程序，而且你又想事半功倍地完成工作，可仔细观察单项贷款利率报价实例（`LoanRateQuote1` 至 `N`）会遇到哪些问题，以及怎样解决这些问题。首先，每个监督者都必须重写通过继承得来的、默认的 `supervisorStrategy` 属性。

```
class LoanRateQuotes extends Actor {

  override val supervisorStrategy =
    OneForOneStrategy(
      maxNrOfRetries = 5,
      withinTimeRange = 1 minute) {
```

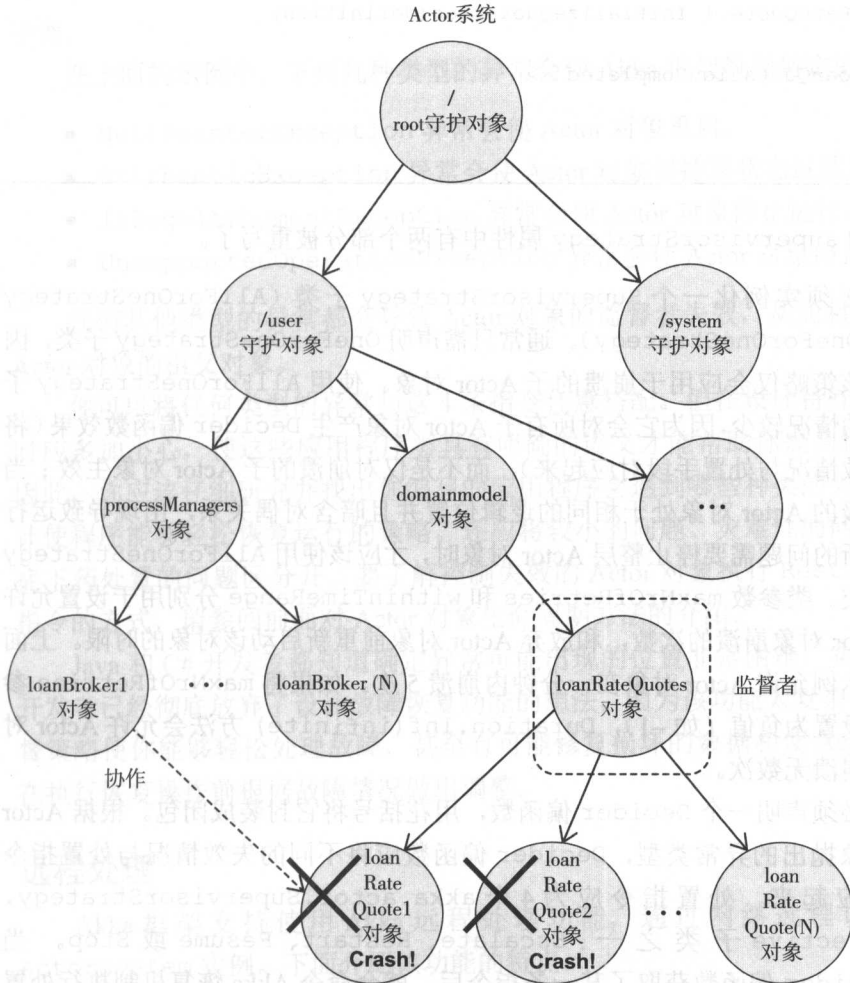


图 2.2 两个 Actor 对象已经崩溃：loanRateQuote1 和 loanRateQuote2。它们的监督者是 loanRateQuotes 对象，而不是 loanBroker1 对象。

```

case NullPointerException => Restart
case ArithmeticException => Resume
case IllegalArgumentException => Stop
case UnsupportedOperationException => Stop
case Exception => Escalate
}
...
def receive = {
  case RequestLoanQuotation(definition) =>
    val loadRateQuote =
      context.actorOf(Props[LoanRateQuote], nextName)

```

```

    loadRateQuote ! InitializeQuotation(definition)
    ...
    case LoanQuotationCompleted =>
    ...
  }
}

```

默认的 `supervisorStrategy` 属性中有两个部分被重写了。

- 你必须实例化一个 `SupervisorStrategy` 子类 (`AllForOneStrategy` 或 `OneForOneStrategy`)。通常只需声明 `OneForOneStrategy` 子类, 因为该策略仅会应用于崩溃的子 Actor 对象。使用 `AllForOneStrategy` 子类的情况较少, 因为它会对所有子 Actor 对象产生 Decider 偏函数效果 (将失效情况与处置手段对应起来), 而不是仅对崩溃的子 Actor 对象生效; 当同级的 Actor 对象处于相同的逻辑位置并且暗含对偶关系, 出现导致运行中断的问题需要停止整层 Actor 对象时, 才应该使用 `AllForOneStrategy` 子类。类参数 `maxNrOfRetries` 和 `withinTimeRange` 分别用于设置允许 Actor 对象崩溃的次数, 和放弃 Actor 对象前重新启动该对象的时限。上面的示例允许 Actor 对象在一分钟内崩溃 5 次。如果将 `maxNrOfRetries` 参数设置为负值 (如 -1), `Duration.Inf(infinite)` 方法会允许 Actor 对象崩溃无数次。
- 你必须声明一个 Decider 偏函数, 用花括号将它封装成闭包。根据 Actor 对象抛出的异常类型, Decider 偏函数应将不同的失效情况与处置指令对应起来。处置指令应为 4 个 `akka.actor.SupervisorStrategy.Directive` 子类之一: `Escalate`、`Restart`、`Resume` 或 `Stop`。当 Decider 偏函数获取了某一条指令后, 就会命令 Akka 恢复机制执行处置操作。大多数指令都很易于理解。`Escalate` 指令的作用是使 Actor 对象的监督者失效, 从而将处置权上交给更高级的监督者 (即 Actor 对象的祖父对象)。`Resume` 指令与 `Restart` 指令不同, `Resume` 指令会完整保留 Actor 对象的状态, Actor 对象能够处理它收到的下一条消息。`Restart` 指令会停止 Actor 对象并彻底重建它, 这意味着会清空 Actor 对象的消息缓存; 该对象与原来的 Actor 对象已经完全不同。

需要批量和单独处理的失效情况都可能会出现。例如, 如果某个数据库查询操作失败了, 就应处理尝试执行该操作的那个 Actor 对象, 即应使用 `OneForOneStrategy` 子类。另一方面, 由于某种原因整个数据库都无法访问, 就应停止执行数据库查询操作的整层 Actor 对象, 即使用 `AllForOneStrategy`

子类。

在上面的示例中，下列几种类型的异常会使 Akka 框架执行特定的恢复操作：

- `NullPointerException` 异常会使 Actor 对象重启。
- `ArithmeticException` 异常会使 Actor 对象保持原状态继续运行。
- `IllegalArgumentException` 异常会使 Actor 对象停止运行。
- `UnsupportedOperationException` 异常会使 Actor 对象停止运行。

任何其他类型的异常都会导致 Actor 对象的监督者失效，从而将处置权交给 Actor 对象的祖父对象。

你可以将任何类型的异常与这 4 条指令任意搭配。但在设计特殊类型的异常时应多加小心，使这些应用程序类具有明确的语义才能帮助你提高程序的韧性。因此，你应该仔细研究并找出你编写的应用程序会遇到哪些种类的失效情况，设计使程序能够轻松恢复运行的策略。还应将较小的问题、灾难性的问题与必须对症下药处置的问题区分开。要了解控制失效的 Actor 对象执行 Restart 和 Stop 指令的方式，请参阅前面对 Actor 对象生命周期方法的介绍。

Java 和 C# 开发者都知道确定异常可能出现的位置非常困难。实际上，有些开发者已经彻底放弃了设计故障恢复功能的想法，因为该功能太复杂了。Akka 监督策略使你能够轻松处理故障，甚至有可能修复损坏的数据和使 Actor 对象能够在执行恢复操作前根据故障情况做出调整。

远程处理

Akka 框架支持使用 Akka 远程处理功能，通过网络远程访问不同的 ActorSystem 实例。下面介绍该功能的特点。

该功能的通信模式是对等网络。通过发送消息，一台 JVM 中的 Actor 对象能与另一台 JVM 中的 Actor 对象直接⁷通信。专门支持非 Actor 客户端与 Actor 服务器进行远程通信的 API 不存在。也许你会将不含有业务逻辑的 JVM 视为客户端。尽管这个想法有一定道理，但这个客户端中必须含有 ActorSystem 系统。客户端中的 Actor 对象能够与你视为服务器的远程 JVM 中的 Actor 对象（这些 Actor 对象也存在于它们本身的 ActorSystem 系统中）通信。更为重要的是，服务器中的 Actor 对象能够与客户端中的 Actor 对象通信。因此，不论被你视为服务器

⁷ 此处的“直接”是指在 Actor 模型中直接通过异步方式传递消息。实际上，该操作不是直接执行的，尤其是在使用网络传递消息的情况中。要将消息送到目标 Actor 对象需要花相当长的时间。然而，从语义方面看，这就是直接通过异步方式执行的消息传递操作。

的 JVM 还是被你视为客户端的 JVM，都具有相同的网络地位，并且都具有双向通信的能力。重新为这两个网络角色下定义可以取得更好的效果，可以将一个 ActorSystem 系统称为用户体验系统，将另一个 ActorSystem 系统称为业务功能系统。显然，你应该为 ActorSystem 对象设计与你的业务符合的名称。

实际上，一个 Actor 系统可以存在于多个节点中。换言之，不必为处于不同 JVM 中的每个网络节点的 ActorSystem 对象起不同的名称。Actor 对象的 Actor 路径中包含了主机和 ActorSystem 对象的名称，因而能够轻松避免 Actor 对象命名冲突。只有在同一台主机和同一个 ActorSystem 系统中，使用已存在的 Actor 对象名称命名新的 Actor 对象，才会导致 Actor 对象命名冲突。因此，两个或多个节点可以使用相同的名称命名本身的 ActorSystem 对象，从而能够使一个 ActorSystem 系统存在于多个节点中。即便如此，还是应该根据 ActorSystem 实例在应用程序中所起的作用，使用不同的名称命名不同的 Actor 系统。

前面的内容已经暗示了访问 Actor 对象的操作具有位置透明性。只要 Actor 对象接入了网络，它们就像都存在于一台大型的 JVM 中一样。下面是简化的本地和远程 Actor 对象的消息传输协议：

```
val someActor: ActorRef = ...
```

```
someActor ! SomeMessage(...)
```

发送者不会知道目标 Actor 对象（不论它是远程 Actor 对象还是本地 Actor 对象）的引用，因为 ActorRef 对象抽象化了这些细节。

使用网络协议（如传输控制协议 TCP），可以使 Actor 对象在不同 JVM 之间进行通信。在这种情况下，每台 JVM 都拥有其本身的 ActorSystem 实例和 Actor 对象。人们通常会认为每台 JVM 都是在物理意义上的不同服务器上运行的，但多台 JVM 可能会在物理意义上的同一台服务器上运行，而且很可能是在同一台虚拟服务器上运行的。无论如何，Actor 对象之间的远程通信都会通过网络在各个 Actor 系统之间实现。

通过网络传递消息，使 Akka 框架拥有了强大的远程功能，该功能使分布式系统可以跨越多个服务器节点。另一方面，在使用网络时，还必须注意随网络而来的各种问题。带宽限制和延迟问题应牢记心间，有时大量的延迟会为网络程序设计提供帮助。然而，你必须考虑的事情不尽于此。

当一条消息经过网络从一个 Actor 对象发送给另一个 Actor 对象时，消息本身一定会被序列化。该操作本身会增加系统开销，根据你选择的序列化类型，该系统开销可能会相当大。默认情况下，Akka 框架使用 Java 序列化，这种序列化

操作在性能和尺寸方面都较差。因此，当你知道消息必须跨越多台 JVM 时，应因地制宜地选择适当的序列化操作。Protocol Buffers[ProtoBuf] 和 Kryo[Kryo] 是比较著名的序列化操作。可以配置你喜欢的序列化类型，并禁用 Java 序列化操作。

```
# application.conf for ActorSystem: RiskRover
akka {
  actor {
    serializers {
      proto = "akka.remote.serialization.ProtoBufSerializer"
      kryo = "com.romix.akka.serialization.kryo.KryoSerializer"
      ...
    }
    serialization-bindings {
      "java.io.Serializable" = none
      ...
    }
    kryo {
      type = "graph"
      idstrategy = "incremental"
      serializer-pool-size = 16
      buffer-size = 4096
      max-buffer-size = -1
      ...
    }
  }
}
```

要在一台 JVM 中指定的 ActorSystem 系统中支持 Akka 远程处理功能，至少要在该对象的 application.conf 配置文件中添加下列配置：

```
# application.conf for ActorSystem: RiskRover
akka {
  ...
  actor {
    # default is: "akka.actor.LocalActorRefProvider"

    provider = "akka.remote.RemoteActorRefProvider"
    ...
  }
  remote {
    # actors at: akka.tcp://RiskRover@hounddog:2552/user
    enabled-transports = ["akka.remote.netty.tcp"]
  }
}
```

```

netty.tcp {
  hostname = "hounddog"
  port = 2552
}
}
}

```

尽管默认情况下使用的配置是 `LocalActorRefProvider`，但是也需要使用 `RemoteActorRefProvider` 配置确保从各种 JVM 引用的 Actor 对象含有远程访问信息。然而，这看起来会增加许多系统开销，如果以处理远程 Actor 对象的方式处理所有本地 Actor 对象，那么在向本地 Actor 对象发送消息时，就必定会空耗网络资源。幸运的是，`RemoteActorRefProvider` 配置不是以这种方式工作的。如果创建的是本地 Actor 对象，实际上远程 ActorRef 引用供应方法（provider）在其内部会使用 `LocalActorRefProvider` 配置。换言之，使用 `RemoteActorRefProvider` 配置的供应方法必须能够分辨出请求获得 Actor 对象的系统是本地的还是远程的，并为它们创建适当的 ActorRef 引用。因此，在供应方法的内部同时存在了 `LocalActorRefProvider` 配置和 `RemoteActorRefProvider` 配置，你需要做的仅是了解 ActorRef 抽象并使用它。

学习了远程处理方式的基础知识后，让我们观察下面两种远程处理方式。

- **远程创建**：通过本地 Actor 系统中的 Actor 对象可以在远程 Actor 系统中创建子 Actor 对象，如图 2.3 所示。我认为这种方式最适合处理负载均衡需求。本地系统中的 Actor 对象知道哪些工作必须完成，但是它没有选择在本地系统中创建子 Actor 对象完成这些工作。这样做的原因可能是在本地节点完成这些工作有过多的系统开销，也可能是由于逻辑层角色的原因本地节点无法完成特定类型的工作。本地 Actor 对象选择将工作分派给独立的远程节点。因此，本地 Actor 对象会在远程节点中创建子 Actor 对象并向该子对象发送工作消息。即使这个子 Actor 对象是在完全不同的系统中创建的，它的所有者也是本地 Actor 对象，而且本地 Actor 对象还是它的监督者。因此，当工作完成后，本地 Actor 对象会负责停止它所有的远程子 Actor 对象。本书会介绍两种远程创建 Actor 对象的方式。
- **远程查询**：本地系统中的 Actor 对象查找远程系统中某个 Actor 对象，如图 2.4 所示。被查询的 Actor 对象是由远程系统中的其他 Actor 对象创建的。本地的 Actor 对象既不是远程 Actor 对象的所有者，也不是远程 Actor 对象的监督者。然而，本地对象可以查询一个或多个远程 Actor 对象，并向它们委派工作。这有点像客户端—服务器模式，因为本地 Actor 对象无法

管理远程 Actor 对象的生命周期，也无法监督远程 Actor 对象。本地 Actor 对象只能使用远程 Actor 对象提供的服务。然而，这不是一种传统的客户端—服务器模式，因为本地 Actor 对象和远程 Actor 对象的网络地位是对等的。

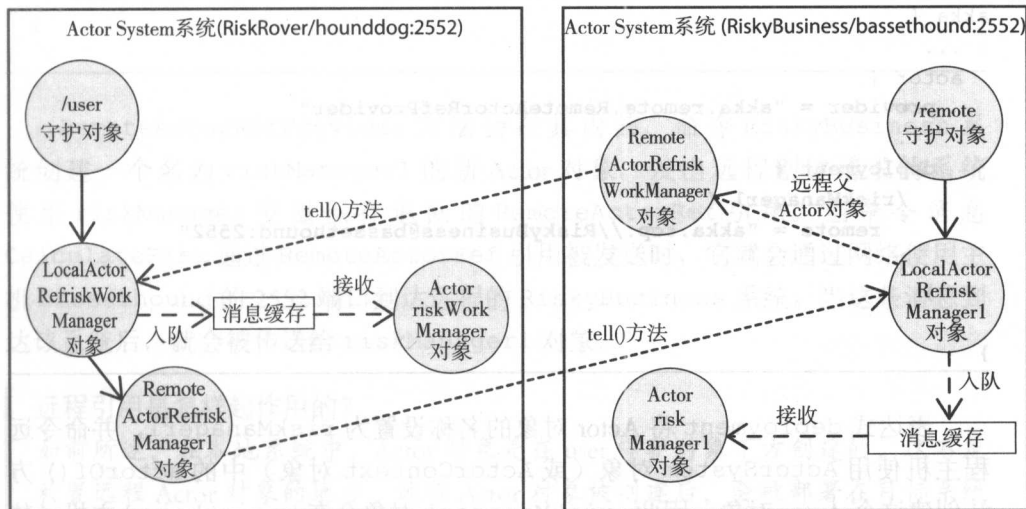


图 2.3 创建远程 Actor 对象的操作会用到 RemoteActorRef 和 LocalActorRef 实例。

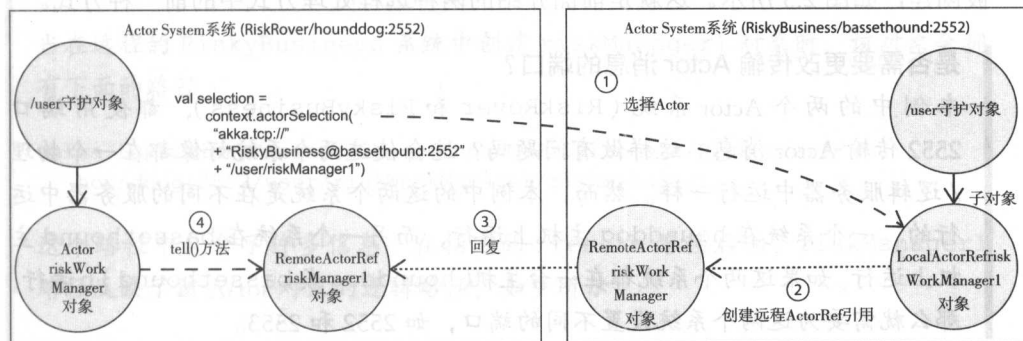


图 2.4 远程 Actor 对象查询是一种异步操作，如果一个或多个远程 Actor 对象符合查询条件，那么该操作最终会在远程系统找到这些符合条件的 Actor 对象。

你应该仔细观察这两种远程处理方式，确定哪种方式最适合你的需求。这两种处理方式很可能都适合你编写的应用程序。

远程创建

在通过本地 Actor 对象创建远程 Actor 对象时，使用配置文件是最简捷的方式。本节会详细介绍这些内容，我们会在名为 basethound 的服务器上创建一

个名为 `riskManager1` 的远程对象。要做到这一点，首先应在 Akka 配置文件 `application.conf` 中添加下面的代码：

```
# application.conf for ActorSystem: RiskRover
akka {
  ...
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
    ...
    deployment {
      /riskManager1 {
        remote = "akka.tcp://RiskyBusiness@bassethound:2552"
      }
    }
  }
}
```

表达式 `deployment` 将 Actor 对象的名称设置为 `riskManager1`，并命令远程主机使用 ActorSystem 对象（或 ActorContext 对象）中的 `actorOf()` 方法创建这个 Actor 对象。因此，`riskManager1` 对象会在 `bassethound` 主机（传输 Actor 消息的端口为 2552）中名为 `RiskyBusiness` 的 ActorSystem 系统中被创建，如图 2.3 所示。这就是前面介绍的两种远程处理方式中的前一种方式。

是否需要更改传输 Actor 消息的端口？

本例中的两个 Actor 系统（`RiskRover` 和 `RiskyBusiness`），都使用端口 2552 传输 Actor 消息。这样做有问题吗？这会使这两个系统好像都在一个物理 / 逻辑服务器中运行一样。然而，本例中的这两个系统是在不同的服务器中运行的。一个系统在 `hounddog` 主机上运行，而另一个系统在 `bassethound` 主机上运行。如果这两个系统都在一台主机（`hounddog` 或 `bassethound`）上运行，那么就需要为这两个系统设置不同的端口，如 2552 和 2553。

下面的代码使用本地 Actor 对象创建了指定的远程 Actor 对象 `RiskWorkManager`，如图 2.3 所示。

```
// RiskWorkManager 对象被部署在主机 hounddog 上（使用 2552 端口传输 Actor 消息）

class RiskWorkManager extends Actor {

  val riskManager =
    context.system.actorOf(Props[RiskManager], "riskManager1")
```

```
def receive = {  
  ...  
  riskManager ! CalculateRisk(...)   
  ...  
}
```

RemoteActorRefProvider 方法会在其内部，命令 RiskyBusiness 系统创建一个名为 riskManager1 的新 Actor 对象。发出远程创建请求的系统使用 riskManager 变量保存返回的 RemoteActorRef 引用。当命令消息 CalculateRisk 通过 RemoteActorRef 引用被发送时，它就会通过网络使用主机 bassethound 的 2552 端口到达远程的 RiskyBusiness 系统。当这条消息到达该系统后，就会被传送给 riskManager1 对象。

远程引用是怎样起作用的？

如前所述，在本地系统中，Actor 对象是在 user 守护对象下方创建的，这不是放置远程 Actor 对象的地方。远程 Actor 对象被创建后，会被部署在目标系统的特殊路径（伪守护者对象 remote）中。就像 user 守护对象一样，remote 守护对象也位于 Actor 层次结构的最顶层。

当在远程的 RiskyBusiness 系统中创建 riskManager1 对象时，该对象会拥有下面的路径：

```
akka.tcp//RiskyBusiness@bassethound:2552/remote/RiskRover@  
hounddog:2552/user/riskWorkManager/riskManager1
```

这个路径中包含了许多信息。在指定的 /remote 路径后面，riskManager1 路径反映了该 Actor 对象的逻辑路径，如下所示：

```
akka.tcp//RiskRover@hounddog:2552/user/riskWorkManager/riskManager1
```

这使 RiskyBusiness 系统所在的主机能够知道怎样为 riskManager1 对象创建 RemoteActorRef 引用，这样才能向对该对象发送消息的 Actor 对象回复消息。不必将 akka.tcp 协议部分保存两次，两端都会使用这部分相同的路径。

因为远程系统拥有了 riskManager1 对象的路径，所以能够轻松推导出其父对象的路径：

```
akka.tcp//RiskRover@hounddog:2552/user/riskWorkManager
```

因此,其他 Actor 对象可以使用在主机 bassethound 上创建的 riskManager1 对象的 RemoteActorRef 引用,通过 riskManager1 对象的父方法,向 riskManager1 对象的父对象(riskWorkManager)发送消息。即使 riskWorkManager 没有被部署在 bassethound 主机上,也可以做到这一点。

为了使远程创建操作能够正确执行,两个系统必须都能够使用为 Actor 类 RiskManager 生成的 Java 类。换言之,RiskRover 和 RiskyBusiness 系统必须能够加载 RiskManager 类。这意味着当前无法在这两个系统(RiskRover 和 RiskyBusiness)之间传输类文件的字节。

此外,CalculateRisk 之类的消息必须能够被序列化,只有这样才能通过网络传输这类消息的字节。如果某个消息类无法被序列化,那么这种消息就无法向远程系统发送。

我们已经在远程系统中创建了一个 Actor 对象并向它发送了一条消息。然而,这条远程通信消息中仍旧缺少了部分信息。前面介绍过,每个 Actor 对象收到消息时,也会获得消息发送者的 ActorRef 引用。只有获得了该引用,收到消息的 Actor 对象才能回复发送者,请参考请求—回复模式。在使用 riskManager1 对象的例子中,当该对象收到 CalculateRisk 消息时,发送者会由 RemoteActorRef 引用代表。这个远程引用会通过 2552 端口指向主机 hounddog 中的远程 RiskRover 系统中的 RiskWorkManager 实例。当 riskManager1 对象回复消息时,它发送的 RiskCalculated 消息会在网络中传输,最终到达远程的 RiskRover 系统。

// riskManager1 对象被部署在主机 bassethound (使用端口 2552 传输 Actor 消息) 上

```
class RiskManager extends Actor {
  ...
  def receive = {
    ...
    senderFor(riskWorkManagerId) ! RiskCalculated(...)
    ...
  }

  def senderFor(id: String): ActorRef = {
    ...
  }
}
```

因为在传输命令消息 CalculateRisk 时一定会存在网络延迟,所以

RiskManager 对象必须对迟到的消息做出回应，同时查明这条消息的发送者（某个 RiskWorkManager 实例）。通过内部方法 senderFor() 传递相关标识符 riskWorkManagerId，可以做到这一点。这个 riskWorkManagerId 标识符必须被包含在该处理过程被发送的各种消息中。一旦获得了这个引用 (RemoteActorRef)，RiskManager 对象就能够向远程系统中的 RiskWorkManager 对象回复 RiskCalculated 事件消息。

你可能会好奇，是否只能直接在指定 Actor 系统的最顶层 (/remote，请参阅前面的补充资料《远程引用是怎样起作用的？》) 下方创建远程 Actor 对象？答案是肯定的，远程创建的 Actor 对象只能位于 /remote 守护对象的下方。

这个限制还有一些其他含义。首先，不应在 /remote 守护对象下方创建过多的 Actor 对象，这会导致可伸缩性问题，因为被扫描对象的数量越多，花费就会越多。其次，通过远程方式创建的最佳 Actor 类型是监工型 Actor 对象，如 riskManager1。设计远程监工对象（如 riskManager1）可以为监工对象创建本地的子 Actor 对象。监工对象可以快速地将工作委派给它的工人子对象。当父对象（如 riskWorkManager）停止了它的远程子对象（如 riskManager1），那么远程子对象的所有子对象（如 riskWorkManager 对象的所有孙对象）也都会被停止。

另一方面，还可以通过另一种方式使用 Actor 对象的远程创建操作：以远程方式创建本地对象的所有子对象。在 application.conf 文件中使用 deployment 表达式可以支持该功能。

```
# application.conf for ActorSystem: RiskyBusiness, RiskRover
akka {
  ...
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
    ...
    deployment {
      /riskManager1 {
        remote = "akka.tcp://RiskyBusiness@bassethound:2552"
      }
      /riskManager1/* {
        remote = "akka.tcp://RiskCalculators@bassethound:2553"
      }
    }
  }
}
```

通过这个 application.conf 文件我们可以看明白一个应用程序中的所

有 ActorSystem 实例共用一个配置文件的方式。不论 riskManager1 对象是在哪个节点中被创建的，/riskManager1 表达式永远会使 riskManager1 对象部署到 RiskyBusiness@bassethound:2552（即使用 2552 端口通信的 bassethound 主机的 RiskyBusiness 系统中）。与此类似，表达式 /riskManager1/* 只有在 riskManager1 对象创建子对象时才会应用于 riskManager1 对象。因此，当 riskManager1 对象创建子对象时，riskManager1 对象就会被部署到 RiskCalculators@bassethound:2553。下面代码为 riskManager1 对象创建了 riskCalculator1 等子对象：

```
// riskManager1 对象被部署到使用 2552 端口通信的 bassethound 主机上
```

```
class RiskManager extends Actor {

    val riskCalculator1 =
        context.system.actorOf(Props[RiskCalculator],
                                "riskCalculator1")

    val riskCalculator2 =
        context.system.actorOf(Props[RiskCalculator],
                                "riskCalculator2")

    val riskCalculator3 =
        context.system.actorOf(Props[RiskCalculator],
                                "riskCalculator3")

    ...
}
```

新建的 Actor 子对象（riskCalculator1、riskCalculator2 和 riskCalculator3）都是在同一台物理主机（bassethound）中创建的，但它们分别处于不同的 JVM 中，使用端口 2553 可以与这些 JVM 通信。这进一步说明当一台物理计算机中的多个独立的 JVM 中含有多个 ActorSystem 实例时，这些 ActorSystem 实例就必须使用不同的网络端口。图 2.5 展示了部署 RiskRover、RiskyBusiness 和 RiskCalculators 系统的详细情况。

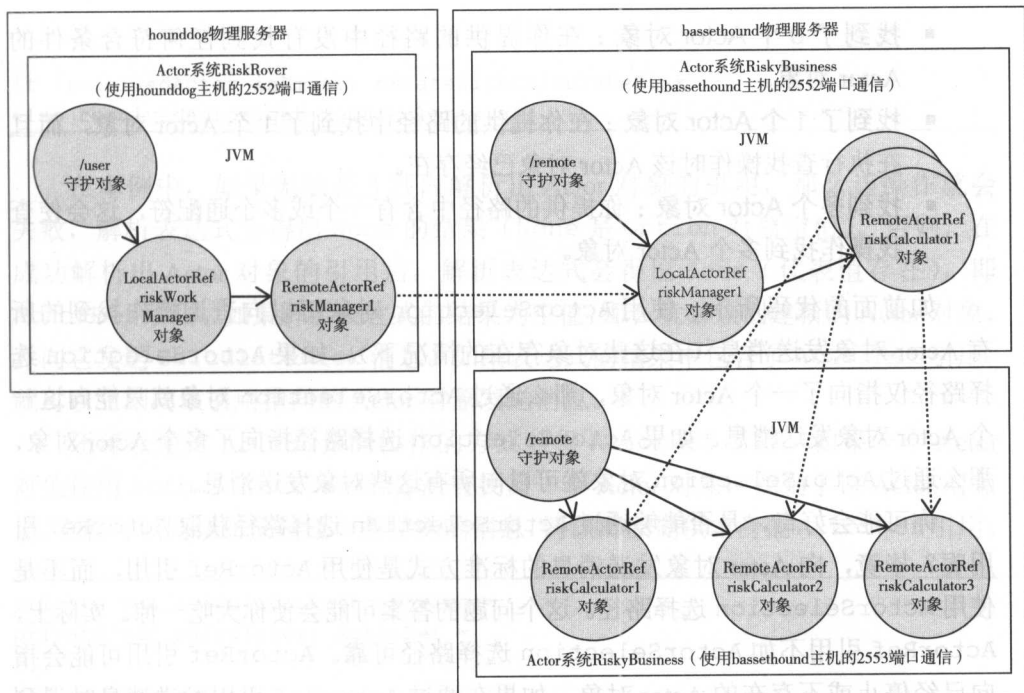


图 2.5 两个物理服务器中含有 3 个部署了 Actor 系统的 JVM。

远程查询

使用远程 Actor 对象的第二种方式是在指定的系统中查询 Actor 对象。在 Akka 框架中这种操作被称为 Actor 选择，因为查询一个或多个 Actor 对象的操作会选中多个 Actor 对象。这部分内容比创建远程 Actor 对象的内容要少很多。如果你要查询的 Actor 对象正在运行，可使用下面的简单方式：

```
import akka.actor.ActorSelection
...
val path = "akka.tcp://RiskyBusiness@bassethound:2552/user/"
riskManager1"
val selection = context.system.actorSelection(path)
selection ! CalculateRisk(...)
```

返回结果的类型为 ActorSelection。一个 ActorSelection 对象可以代表 0 个、1 个或多个 Actor 对象，这取决于你为 actorSelection() 方法提供的路径类型。

- 找到了 0 个 Actor 对象：在你提供的路径中没有找到任何符合条件的 Actor 对象。
- 找到了 1 个 Actor 对象：在你提供的路径中找到了 1 个 Actor 对象，并且在执行查找操作时该 Actor 对象已经存在。
- 找到多个 Actor 对象：你提供的路径中含有一个或多个通配符，这会使查找操作找到多个 Actor 对象。

如前面的代码所示，使用 ActorSelection 对象可以向查询操作找到的所有 Actor 对象发送消息（在这些对象存在的情况下）。如果 ActorSelection 选择路径仅指向了一个 Actor 对象，那么通过 ActorSelection 对象就只能向这一个 Actor 对象发送消息。如果 ActorSelection 选择路径指向了多个 Actor 对象，那么通过 ActorSelection 对象就可以向所有这些对象发送消息。

你可能会好奇，是否能够通过 ActorSelection 选择路径获取 ActorRef 引用呢？毕竟，向 Actor 对象发送消息的标准方式是使用 ActorRef 引用，而不是使用 ActorSelection 选择路径。这个问题的答案可能会使你大吃一惊。实际上，ActorRef 引用不如 ActorSelection 选择路径可靠。ActorRef 引用可能会指向已经停止或不存在的 Actor 对象。如果在通过 ActorRef 引用发送消息时遇到了这些情况，那么消息就无法被送到因为没有接收消息的 Actor 对象。另一方面，ActorSelection 选择路径是以 ActorPath 路径为基础的。通过路径发送消息的方式会使 Akka 框架在内部通过路径查找 Actor 对象，并在找到指定的 Actor 对象后向其发送消息。那么这其中有哪些区别呢？其中的区别在于，当 Actor 对象重启后它的 ActorRef 引用会改变，但是它的 Actor 系统路径不会改变。

即便如此，如果你更喜欢使用引用，也可以通过迂回的方式使用 ActorSelection 选择路径获得 ActorRef 引用。你可以使用 resolveOne() 方法，通过 ActorSelection 路径解析出单个的 ActorRef 引用。

```
import akka.actor.ActorSelection
...
val path = "akka.tcp://RiskyBusiness@bassethound:2552/user/riskManager1"

val selection = context.system.actorSelection(path)

val resolvedActor =
  selection.resolveOne(Timeout(3000)).onComplete {
    case Success(resolved) => Some(resolved)
    case Failure(e) => None
```

```
}
if (resolvedActor isEmpty) createWithCalculateRisk
else resolvedActor.get ! CalculateRisk(...)
```

在本例中，如果无法在 3 秒内解析出 Actor 对象的引用，那么该操作就会失败，解析表达式会得出 None 的结果（None 是 Option 对象空值）。否则，在成功解析出 Actor 对象的引用后，解析表达式会得出 Some（代表值存在），即 ActorRef 引用。如果解析表达式的结果为空值，那么就必须创建新的 Actor 对象，并向它发送 CalculateRisk 消息。如果表达式的结果中包含了 ActorRef 引用，就可以立刻向该引用指向的 Actor 对象发送消息。

还可以通过 Actor 对象间协作的方式获得 ActorRef 引用。如果一个 Actor 对象使用 actorSelection() 方法查找另一个 Actor 对象，找到了该 Actor 对象后，它就可以向该对象发送一条特殊的消息，请该对象提供本身的 ActorRef 引用。

```
private var calculatorPath: String = _
private var calculator: ActorRef = _
...
def receive = {
  case CalculateUsing(searchPath) =>
    val selection = context.actorSelection(searchPath)
    selection ! Identify(searchPath)
    calculatorPath = searchPath
  case identity: ActorIdentity =>
    if (identity.correlationId.equals(calculatorPath) {
      calculator = identity.ref.get
      calculator ! CalculateRisk(...)
```

在本例中，一个充当路由器的 Actor 对象收到了一条 CalculateUsing 消息，并调用 actorSelection() 寻找担任风险计算器的 Actor 对象。获得该对象的 ActorSelection 选择路径后，充当路由器的 Actor 对象会向找到的 Actor 对象，发送带有 searchPath 参数的 Identify 消息。该参数可以是任意值，但该值必须能够唯一标识这条消息，并能够将恢复的消息关联起来。最后，当充当路由器的 Actor 对象收到 ActorIdentity 消息后，会使用已经保存的 calculatorPath 路径检查指定的 correlationId 对象。如果它们指向的是同一个 Actor 对象，就说明这个解析操作成功完成了，充当路由器的 Actor 对象会将通过 ActorIdentity 消息获得的 ActorRef 引用保存到 calculator 变量中。这个就是我们要查找的担任风险计算器的 Actor 对象的 ActorRef 引用。

Akka 框架的集群功能是以远程处理功能为基础的。尽管 Akka 集群功能可以满足应用程序的所有需求，但是在某些特殊情况中使用 Akka 远程处理功能会更加方便。下面介绍 Akka 框架的集群功能。

集群功能

这部分内容由 Will Sargent 撰写

Akka 集群功能具有 Akka 远程处理功能所没有的优势，使用 Akka 集群功能时不必知道任何资源的具体位置。通过使用 Akka 集群功能，能够在程序正在运行的时候将主机添加到集群中，也能够将主机从集群中撤出。在需要以可伸缩方式使用资源的情况中，该功能极具吸引力。通过使用 Akka 集群功能，当需求高峰来临时，可以向集群中供应、添加更多主机，将这些主机投入工作之中，当负载降低集群中不需要过多的资源时，可以从集群中撤出多余的主机。

要创建能够使用集群功能的 Actor 对象，需要使用下面的工具，稍后会更详细地介绍它们。

- **集群单例对象**：在集群中仅存在一个实例的 Actor 对象。
- **集群分片**：Actor 对象能够均匀地分布在集群的所有节点中。
- **集群客户端**：集群外部的 Actor 对象能够在不知道集群内部 Actor 对象的具体位置的情况下，向集群内部的 Actor 对象发送消息。
- **集群路由器**：集群中的各个节点使用路由器分派工作。

Akka 集群功能不仅处理需求高峰问题，还可以用于故障转移。即使你仅拥有 5 台计算机，通过 Akka 集群功能在这些计算机之间进行负载均衡，和出现意外故障时将工作转移给正常运行的计算机，也能够提高资源的使用效率。

Akka 集群功能专门用于支持多节点的、容错的、分布式 Actor 系统。它通过创建节点集群来做到这一点。任何一个节点都必须是一个能够使用 TCP 通过某个端口通信的 ActorSystem 对象，只有这样它才能拥有唯一标识。所有节点必须使用相同的名称命名本身的 ActorSystem 系统。同一台主机中的多个节点必须使用不同的端口。在同一台物理主机中的同一个集群中的两个节点不能使用相同的端口。ActorSystem 实例之间不存在全局状态，因此在一台主机中，你可以在一台 JVM 上创建多个节点，也可以在一个节点中包含多台 JVM。

根据具体的扩展和收缩需求，你可以启动任意数量的 JVM 节点，以便取得最佳效果。为了满足应用程序（如含有 2,400 个节点的集群）[2,400-Node Cluster] 的需求，在适当数量的物理服务器主机中启动数千台 JVM，也不会有什么問題。

集群会自动检测新加入的节点和出故障的节点。集群会处理新加入的节点，原来处于集群中的节点也会离开集群，其原因包括需求选择、出了故障和为了在集群的所有节点之间使用一种标准通信协议。Akka 集群节点之间使用的标准通信协议称为 Gossip 协议 [Gossip Protocol]。该协议要求所有节点不断报告它们的可用情况，以便使集群能够维持正常运转。集群中的某个节点起领导者的作用，如果该节点出了故障，另一个节点会接替它成为新的领导者。

集群中的每个节点都需要拥有具有唯一性的标识符。这个唯一标识符含有运行 JVM 的主机的名称、主机的端口号、ActorSystem 实例提供的标识符。下面是集群节点完整的标识符格式：

主机名：端口号：ActorSystem 实例提供的标识符

这 3 个部分组合到一起在集群的生命周期中具有唯一性，并且不能重复使用。这也是不允许独立的 JVM 节点共用端口的原因之一（在现实情况中，即使不将端口号用作唯一标识符的组成部分，让两个节点使用相同的端口也不合适）。此外，当 JVM 节点离开集群时，它可能不会使用原来的 ActorSystem 对象重新加入集群。尽管可以再次使用原来的主机名和端口，但原来的 Actor 系统必须关闭，并启动一个新的 Actor 系统，这个新的 ActorSystem 对象会提供新的唯一标识符。图 2.6 展示了 3 个物理服务器，每台物理服务器中都含有一个集群中的 4 台 JVM。

要使用 Akka 集群功能，至少必须在 build.sbt 文件中添加下面的代码：

```
libraryDependencies ++= Seq("com.typesafe.akka" %% "akka-cluster" % "2.4")
```

还必须在 application.conf 文件中包含这种配置：

```
akka {  
  actor {  
    provider = "akka.cluster.ClusterActorRefProvider"  
  }  
  remote {  
    log-remote-lifecycle-events = off  
    netty.tcp {  
      hostname = "hounddog"  
      port = 0  
    }  
  }  
  cluster {  
    seed-nodes = [  

```



```

    "akka.tcp://RiskRover@hounddog:2551",
    "akka.tcp://RiskRover@bassethound:2551"]

    auto-down-unreachable-after = 10s
  }
}

```

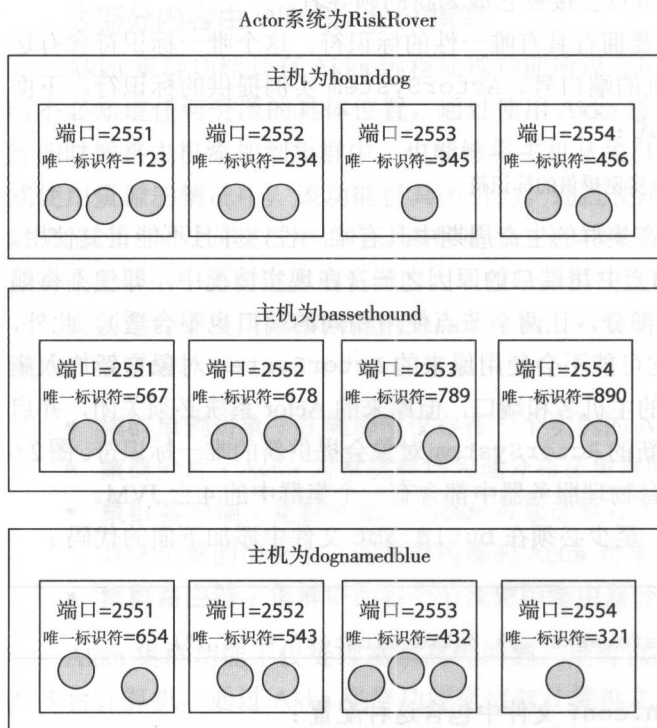


图 2.6 这个 Akka 集群含有 3 台物理服务器，每台物理服务器中运行了 4 台 JVM，其中含有 12 个节点。

在你编写的应用程序中，可能会为种子节点使用不同的端口号或主机名，但是你使用的 `application.conf` 文件必须包含这些基础部分（因为不同的环境中会使用不同的主机名，所以有些程序员会将某个环境专用的集群设置保存在 `cluster-${projenv}.conf` 文件中，并将该文件包含到 `application.conf` 文件中）。

两个种子节点的定义不会限制你的集群只能包含两个节点，该定义仅是指明该集群中至少会存在两个节点，而且在集群启动时这两个节点会起到种子的作用。一个种子节点就像一个大家都认识的熟人，它使其他节点能够加入到集群中。称为种子节点不会使该节点变得特殊；换言之，种子节点不必成为领导者。此外，

新加入的节点不使用种子节点也可以加入集群。它们可以向集群中的其他节点提出加入请求。

还应注意一个特殊情况：在启动集群时，种子节点列表中的第一个节点必须先于其他节点启动。这确保了当第一次初始化集群时，种子节点不会形成多个独立部分。

因为 Akka 集群功能是以 Gossip 协议（以不间断的方式发送确认节点处于正常工作状态的消息）为基础的，所以确保 Actor 系统永远（即使在有负载的情况下）以不间断的方式发送确认节点处于正常工作状态的消息非常重要。如果确认节点处于正常工作状态的消息被延迟得过长，那么集群就会累计这些错误并最终将该节点视为出了故障，并将该节点划出集群。要确保确认节点处于正常工作状态的消息能够被优先传送，应在消息分配器中配置它们，以便使这类消息总是能获得线程。

```
akka.cluster.use-dispatcher = cluster-dispatcher
cluster-dispatcher {
  type = "Dispatcher"
  executor = "fork-join-executor"
  fork-join-executor {
    parallelism-min = 2
    parallelism-max = 4
  }
}
```

加入集群 一个节点可以通过多种方式加入集群。第一种加入集群的方式是通过配置使用已定义的种子节点。

```
val port = 0
val config = ConfigFactory
    .parseString(
      s"akka.remote.netty.tcp.port=$port")
    .withFallback(
      ConfigFactory.parseString(
        "akka.cluster.roles = [frontend]"))
    .withFallback(ConfigFactory.load())

val system = ActorSystem("RiskRover", config)
```

通过这种方式，可以使用在 akka.cluster.seed-nodes 配置中定义的种子节点，创建集群的初始连接。所有必要的信息都包含在 config 对象中。这样做的优点是简捷；如果在配置中写入种子节点的地址，而且有种子节点启动了，那

么就万事大吉了。

然而，如果配置不是被包含在一个文件中（这种情况很可能会出现，例如，集群位于纯云计算中，或者种子节点信息被保存在 Zookeeper 或 Consul 之类的配置服务中），那么使用配置添加节点就不合适。

第二种加入集群的方式是编程方式。如果无法使用种子节点配置系统，那么第一个加入集群的节点就必须加入自己，从而开启向集群添加节点的过程。

```
val address = Cluster(system).selfAddress
Cluster(system).join(address)
```

如果配置系统中有种子节点，那么该节点可以变成通过配置系统定义的种子节点。

```
val seeds = configurationSystem
    .getNodeAddresses().map { nodeAddress =>
        AddressFromURIString(s"akka.tcp://${nodeAddress}") }
Cluster(system).joinSeedNodes(seeds)
```

注意，当调用 `joinSeedNodes()` 方法时，将要加入集群的节点不应出现在该列表中（即该列表不应含有节点本身的地址）。

第三种加入集群的方式是手动方式。你可以使用 Java 管理扩展（Java Management Extensions, JMX）、MBean 类加载器 `akka.Cluster` 或封装了 JMX 的命令行客户端，创建一个节点并将之添加到集群中。只有当通过外部工具管理正在运行的集群时（如在部署和开通集群的情况中），才能够这样做。在某些情况中，操作人员比硬编码更加了解网络拓扑结构，因此会通过执行操作，从系统外部管理集群。

在这种情况下，如果要将指定的节点从一个集群迁移到另一个集群（分别为 `$SEEDNODEHOST` 和 `$SEEDNODEPORT`），可以使用下面的代码：

```
bin/akka-cluster $NODEHOST $JMXPORT is-available
bin/akka-cluster $NODEHOST $JMXPORT member-status
bin/akka-cluster $NODEHOST $JMXPORT leave
    akka.tcp://RiskRover@$NODEHOST:$NODEPORT
bin/akka-cluster $NODEHOST $JMXPORT join
    akka.tcp://RiskRover@$SEEDNODEHOST:$SEEDNODEPORT
bin/akka-cluster $NODEHOST $JMXPORT cluster-status
```

集群角色和事件 每个节点都能够扮演特定的应用程序角色，在这种情况下，角色承担的任务由节点中的 Actor 对象完成。出现多个节点扮演相同角色的可能性很大。如图 2.6 所示，你可以轻松地分辨出有些节点扮演的是风险管理师、

有些节点扮演的是风险计算器，而另一些节点扮演的是风险评估师。你可以使用 `akka.cluster.roles` 配置在 `application.conf` 文件中为每个节点分配角色。

```
akka {
  ...
  cluster {
    roles = [riskWorkManagers]
    ...
  }
}
```

还可以通过命令行界面，使用启动参数为指定的节点分配角色。

```
-Dakka.cluster.roles=[riskCalculators]
```

如前面“加入集群”中的例子所示，指定节点扮演的角色是与 `MemberUp` 事件一起传送的。订阅 `MemberUp` 事件并监视扮演特定角色的新加入节点，就能够使 `Actor` 对象加入特定的工作角色。通过下列方式可以使 `Actor` 对象订阅指定类型的集群事件：

```
cluster.subscribe(
  self,
  classOf[MemberEvent],
  classOf[UnreachableMember])
```

订阅者 `Actor` 对象收到的第一个事件是 `CurrentClusterState`，该事件中含有集群彻底构成后的全部状态。`CurrentClusterState` 事件后面会跟着增量改变事件（如 `MemberUp`，该事件代表新加入的节点彻底启动并能够投入工作了）。下面列出了一些节点事件，表 2.4 详细介绍了它们：

```
class RiskWorkManager extends Actor {
  ...
  def receive = {
    case state: CurrentClusterState =>
    case MemberUp(member: Member) => ...
    case MemberExited(member: Member) => ...
    case MemberRemoved(member: Member) => ...
    case UnreachableMember(member: Member) => ...
    case ReachableMember(member: Member) => ...
  }
}
```

通过调用下面的方法，可以查看集群中节点的状态：

```
val system = ActorSystem("RiskRover")
...
val state = Cluster(system).state
```

表 2.4 Akka 集群功能提供的常用节点事件

事件	描述
CurrentClusterState	这是节点订阅集群事件后收到的第一个事件。如果节点在订阅集群事件时，集群中还没有其他节点，该事件中就不会含有节点信息。否则，该事件会含有 members 集合，该集合中的每个 Member 元素都代表一个集群内的节点。所有 Member 元素都是随独立的节点事件一起发送的，通过这些元素可以查明它们代表的节点扮演什么角色。这些元素包含下列方法：allRoles():Set[String]、leader():Option[Address]、roleLeader():Option[Address] 和 unreachable():Set[Member]
MemberUp	该事件代表新加入集群的节点已经可以投入工作了。使用 Member 参数可以查明新加入节点扮演的角色。使用 hasRole("roleName") 方法可以获得 Boolean 型值；使用 roles 方法可以获得包含集群中节点扮演的所有已命名角色的集合。使用 status 方法可以获得 Joining、Up、Leaving、Exiting 和 Down 等值其中之一。该事件中还含有其他方法，但这些方法都不是应用程序级的
MemberExited	该事件代表集群中的某个节点被关闭了，以便退出集群，而且当收到这个事件时这个节点可能已经退出了集群。只有当退出节点的 Actor 系统被关闭并重启，能够提供新的集群成员标识符后，该节点才能再次在集群中被使用
MemberRemoved	该事件代表集群已经主动移除了某个节点。只有当被移除节点的 Actor 系统被关闭并重启，能够提供新的集群成员标识符后，该节点才能再次在集群中被使用
UnreachableMember	该事件代表集群检测到某个节点出了故障，并将该节点标记为不可用。这可能仅是一种临时状态，该情况也可能会升级为 MemberRemoved 事件。导致节点不可用情况的一个原因是网络拥堵问题，在这种情况下节点无法在周期时限内收到集群中其他节点发送的确认它处于正常工作状态的消息
ReachableMember	该事件代表集群重新将某个节点视为可用节点。节点不可用的一个原因是，因网络拥堵而无法在周期时限内送达确认节点正常运行的消息。当网络情况变好并能够在周期时限内送达确认该节点正常运行的消息时，集群会重新将该节点视为可用的节点

state 变量引用了一个 `CurrentClusterState` 实例，如表 2.4 所示。你必须认识到当收到该事件时，集群的状态可能已经有了新的变化。集群中活动节点的变化速度，可能远快于被视为在当前时间拍摄的状态快照。

集群单例对象 有时你需要使某个 Actor 对象在整个集群中起作用。当某个 Actor 对象在整个集群中的任务始终如一（该对象监听或收集集群中的信息，或者该对象被用作中心节点，为集群中的其他节点转发消息），应使用这种处理方式。使用一个 Actor 对象收集集群中每个节点的权值并广播整个集群的运行情况，就是一个典型的例子。另一个例子是将一个 Actor 对象用作工作路由器，在这种情况下所有工作任务都必须经过一个路由器，这样该路由器就能够控制完成任务的方式并监控它们的完成程度。

通过为 Actor 对象设置专用的 `ClusterSingletonManager` 属性，可以创建集群单例对象。

```
import akka.contrib.pattern.ClusterSingletonManager
...
val clusterSingletonProperties =
  ClusterSingletonManager.props(
    singletonProps = Props(classOf[SingletonActor]),
    singletonName = "SingletonActor",
    terminationMessage = PoisonPill,
    role = None)
system.actorOf(clusterSingletonProperties, "clusterSingleton")
```

这个 Actor 系统会在集群中运行时间最久的节点中创建集群单例对象，不必在提出创建请求的节点上创建集群单例对象。如果含有集群单例对象的节点不能继续工作了，发送给该对象的消息会缓存在本地代理对象中，直到新的集群单例对象被创建为止。

使用集群单例对象时需要注意一个问题：创建集群单例对象的前提是集群还没有分区，因此如果集群中出现了分区并选出了新的领导者，那么很可能需要创建第二个集群单例对象。这种情况通常称为裂脑 [Split-Brain]。避免出现这种情况，可通过设置 `min-nr-of-members` 属性，在集群拥有法定数量节点前，防止进行领导权选举。

集群分片 分片是一种常用的处理方式，用于处理数据平均分布在多个节点中的情况。集群分片使 Actor 对象能够均匀地分布在集群的所有节点中，集群中的每个节点都起一个分片的作用。这一点非常重要，因为 Actor 对象本身只能占用较少的内存，各个节点中数量众多的 Actor 对象占用的内存比一台 JVM 中的 Actor 对象占用的内存多得多。进行分片的另一个原因是建立节点和 Actor 对象之间的紧密关系，这会使得你有计划地在同一个节点中保存有关联的 Actor 对象，

以便使这些对象以本地方式传递消息。进行分片的第三个原因是将特定类型的 Actor 对象集中到集群中某个固定区域。通过使用集群分片，可以使 Actor 对象以透明方式分布在集群中的多个节点中，并且能够通过标识符被引用。

分片中的一个 Actor 对象被称为条目。一个分片能够包含许多条目。图 2.7 展示了一个分片示例。集群单例对象 ShardCoordinator 知道每个分片的位置，并将每个分片与一个区域对应起来。Actor 对象 ShardRegion 在集群中的所有节点上运行，并且含有许多分片。ShardCoordinator 对象监视哪个区域中的分片数量最多，而且会在默认情况下向含有分片最少的区域中分配新的分片。

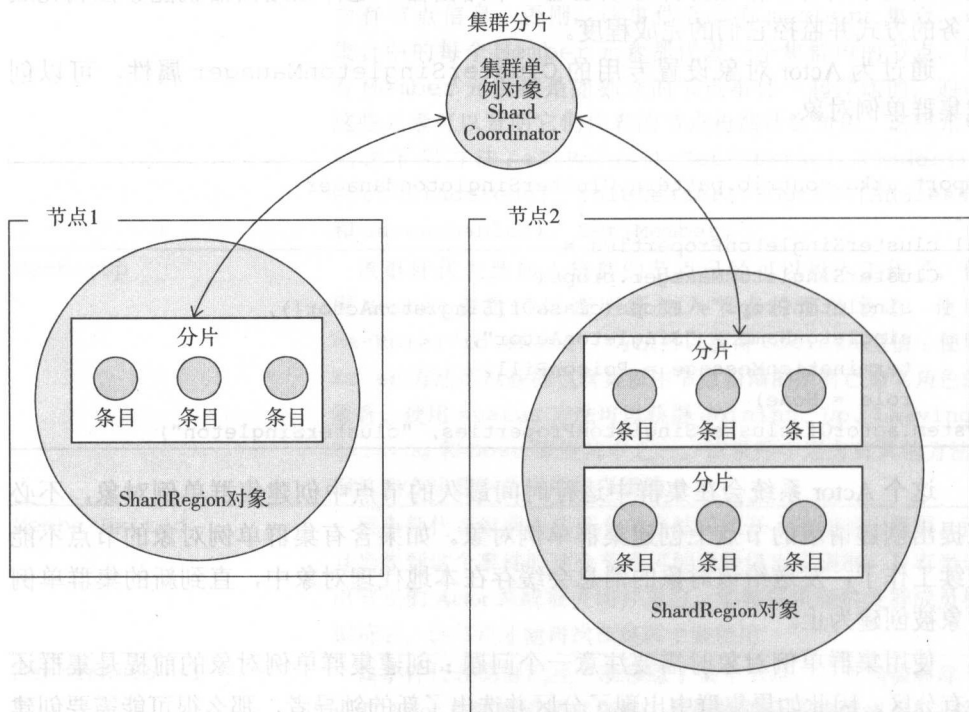


图 2.7 含有 ShardCoordinator、ShardRegion 实例、分片和条目的集群分片示例。

在分片中条目是被怎样放置的？

最初，集群分片中不含有任何条目。当向条目发送消息时，即使该条目没有位于内存中，该条目也会被找到。这意味着这条消息被本应包含该条目的分片检测到了，即使该分片还没有包含该条目。如果该条目还不存在，那么它就会在该消息送达前自动被创建。如果该条目通过扩展 PersistentActor 类拥有了存在前状态，在被创建后消息送达前该条目的状态会被先存储。最终，

这条消息会被传送给检测到它的分片中新创建的或拥有存在前状态的条目。

当条目还没有位于内存中，怎样才能找出该条目并向它发送消息呢？如果该条目曾经存在过，可以通过查询数据存储区找到它。你至少能够拥有一些与该条目的可用于查询的属性匹配的由用户提供的数据。解决了查询问题后，就可以通过 `ShardRegion` 对象向该条目发送消息，然后一切都会自动水到渠成。

所有这些功能使条目能够在内存中自由来去（也称为钝化），并能够存在于各种节点中，你不必考虑它们的运行细节。

当集群中新加入了节点时，集群单例对象就能够重新平衡各个分片的负载，这样分片中原来的 `Actor` 对象就能够从一个节点迁移到另一个节点。`Actor` 对象的内部状态不会随 `Actor` 对象迁移，因此通常应使用 Akka 的 `Persistence` 工具包确保能够恢复 `Actor` 对象的状态，尽管这不是一个硬性要求。集群单例对象 `ShardCoordinator` 本身就是一个 `PersistentActor` 对象，因为它必须随时了解所有分片的位置。这意味着必须使用 Akka 的 `Persistence journal` 插件配置集群分片，如应用于 `Cassandra` 数据库的插件。

分片注意事项

集群分片只能确保将 `Actor` 对象均匀地分布到集群中的节点中。此外，如果某些分片或节点有较重的负载，分片无法解决这个问题。这是因为共用数据的性能和负载问题位于应用程序层面。你能够了解到的仅是哪类数据会比较多和导致这种情况出现的原因。

当一条消息被发送时，会先到达节点中的相关 `ShardRegion` 对象。`ShardRegion` 对象会提取出条目的位置信息，并将这条消息转发给符合条件的 `Actor` 对象，如图 2.8 所示。

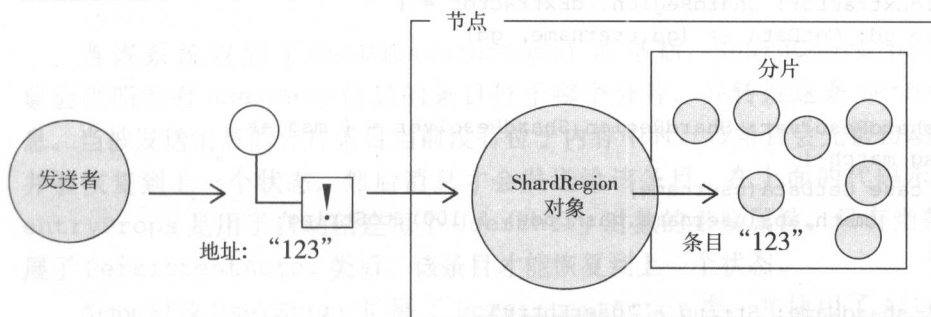


图 2.8 `ShardRegion` 对象会提取消息中目标条目的位置信息，这样它就能够将指定的消息发送给相关的 `Actor` 对象。

消息是以尽力而为方式发送的。集群分片无法确保消息一定会被送到目标对象，而且消息在分布式网络中的传输频度较高。这意味着丢失发送给某条目的消息的可能性也较高，因为默认情况下消息是以至多执行一次的方式被发送的，如图 2.9 所示。AtLeastOnceDelivery 是 Akka 的 Persistence 工具包提供了一种功能，使用该功能可以确保能够成功将消息送达目的地。

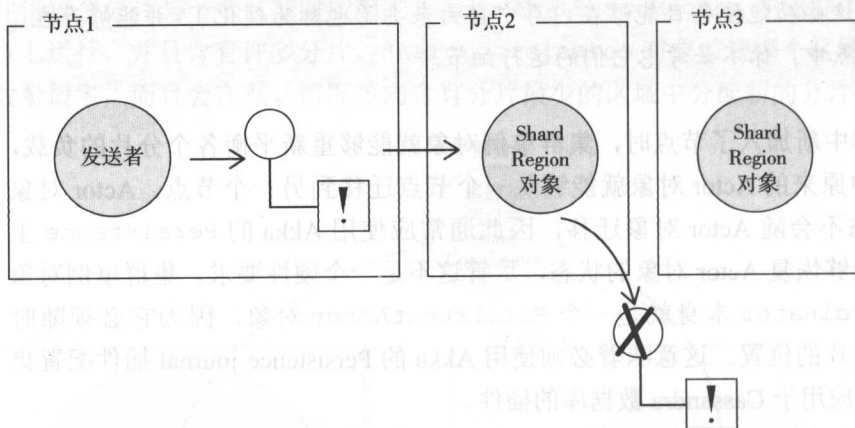


图 2.9 通过至多执行一次的方式，消息在传送时丢失的可能性较高。

集群分片适用于多状态对象能够准确地与 Actor 对象对应起来的情况。例如，可以使用集群分片将应用程序用户建模为 Actor 对象。

```
object UserEntry {
  def props(): Props = Props(new UserEntry())
  case class GetData(username:String)
  case class UserDataAdded(firstName:Option[String],
                           lastName:Option[String])

  val idExtractor: ShardRegion.IdExtractor = {
    case gd: GetData => (gd.username, gd)
  }

  val shardResolver: ShardRegion.ShardResolver = { msg =>
    msg match {
      case GetData(username) =>
        (math.abs(username.hashCode) % 100).toString
    }
  }

  val shardName: String = "UserEntry"
}
```

```
class UserEntry
  extends PersistentActor
  with AtLeastOnceDelivery {
  import UserEntry._

  private var firstName: Option[String] = None
  private var lastName: Option[String] = None

  override def receiveRecover = {
    case UserDataAdded(f, l) =>
      firstName = Option(f)
      lastName = Option(l)
  }

  override def receiveCommand = {
    case AddUserData(f, l) =>
      persist(UserDataAdded(f, l)) { evt =>
        firstName = f
        lastName = l
      }
    case GetData(_) =>
      sender() ! UserData(firstName, lastName)
  }
}
```

通过扩展 ClusterSharding 类可以启动这个集群系统。

```
ClusterSharding(system)
  .start(
    typeName = UserEntry.shardName,
    entryProps = Some(UserEntry.props()),
    idExtractor = UserEntry.idExtractor,
    shardResolver = UserEntry.shardResolver)
```

当该系统收到了 GetData(username) 消息后, ShardCoordinator 对象会查明含有 username 信息的条目位于哪个分片, 并转发这条 GetData 消息。当被发送消息的分片条目当前没有位于内存中时, 该条目会先被自动创建, 并被恢复到上一个状态, 然后消息才会发送给该条目。在上面的代码示例中, entryProps 是用于自动创建每个 UserEntry 对象的必要属性。只有当条目扩展了 PersistentActor 类后, 该条目才能恢复到上一个状态。

Actor 对象 UserEntry 扩展了 PersistentActor 类, 并使用了 AtLeastOnceDelivery 特征, 以便能够从节点故障和丢失消息情况中恢复; 请参阅第 5

章和第7章。

集群客户端 一旦定义了集群后，集群外部的 Actor 对象就能够在不了解集群内部某个 Actor 对象具体位置的情况下，向该 Actor 对象发送消息。在集群内部的 Actor 对象不是集群单例对象和没有通过分片管理集群的情况中，这一点特别重要。这种向集群内部 Actor 对象发送消息的方式是集群客户端。

设置集群客户端需要使用一些辅助工具，这些工具用于管理客户端 Actor 对象与集群内部 Actor 对象之间的通信。使用特殊的 ClusterReceptionist 对象可以做到这一点，在集群内部使用该对象可以注册将会向集群外部的客户端提供服务的 Actor 对象。ClusterReceptionist 是一种普通的 Actor 对象，但是它必须存在于集群中的每个节点中，如图 2.10 所示。通过在 application.conf 文件中设置 ClusterReceptionistExtension 配置，可以在所有节点中自动启动该对象。

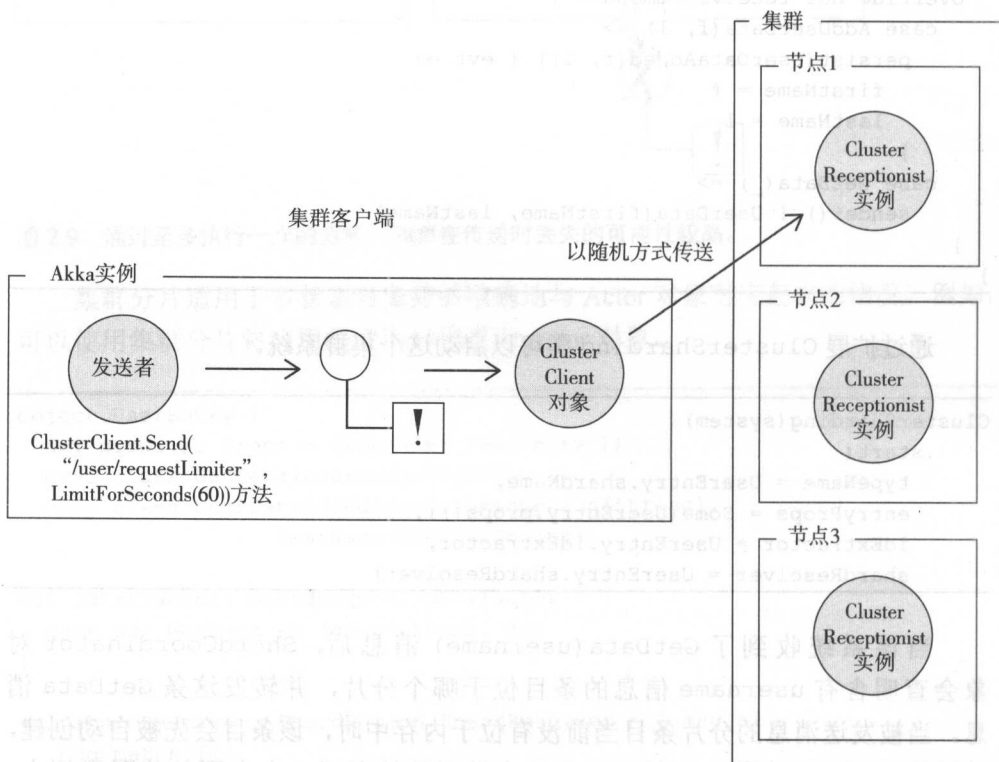


图 2.10 发送者通过 ClusterClient 对象发送了一条消息，该消息会以随机方式被传送给 3 个 ClusterReceptionist 实例其中之一。

```
akka.extensions =
  ["akka.contrib.pattern.ClusterReceptionistExtension"]
```

一旦该集群中担任接待员的对象启动后，集群外部的客户端就能够使用集群内部已经注册的 Actor 对象（不论这些 Actor 对象是在哪些节点上运行）。你可以使 Actor 对象在任意数量的节点上运行。唯一的限定因素是要通过集群外部的客户端使用集群内部的 Actor 对象，就必须使用集群中的接待员对象注册这些 Actor 对象。这样就在每个节点上都安装了一个节点监控器，从而能够监控和禁用集群内部 Actor 对象提供的服务：

```
// 设置能够向集群外部提供服务的所有 Actor 对象
// 通过客户端，将 Actor 对象注册为服务
val requestLimiter: ActorRef =
  system.actorOf(
    Props[RequestLimiter],
    "requestLimiter")

ClusterReceptionistExtension(system)
  .registerService(requestLimiter)
```

顺便提一句，本例中 Actor 对象 RequestLimiter 的基础思路，是使该对象能够接收集群外部的请求，但也能够拒绝处理在某段时间收到的请求。RequestLimiter 对象可以接受 LimitForSeconds(nrOfSeconds) 类型的消息，以便使其能够在它所在的节点中在指定的时间段内拒绝处理消息。

下面介绍通过客户端访问服务的方式。要通过客户端访问服务，就必须设置联络点和初始化集群客户端。可以通过与配置种子节点类似的方式，设置集群中初始的联络员对象。首先，应配置 application.conf 文件中的 contact-points 属性。

```
contact-points = [
  "akka.tcp://RiskRover@host1:2551",
  "akka.tcp://RiskRover@host2:2552"]
```

然后，通过配置将初始的联络点读入集群中的接待员对象。

```
// 设置集群中的接待员节点
val initialContacts =
  immutable.Seq(config.getStringList("contact-points")).map {
    case AddressFromURIString(addr) =>
      system.actorSelection(
        RootActorPath(addr) / "user" / "receptionist")
  }.toSet
```

从逻辑方面讲，这段代码与下面的代码相等：

```

val path1 =
  "akka.tcp://RiskRover@host1:2551/user/receptionist"
val path2 =
  "akka.tcp://RiskRover@host1:2552/user/receptionist"
val initialContacts =
  Set(system.actorSelection(path1),
    system.actorSelection(path2))

```

但使用配置文件的效果更好。通过使用 `initialContacts` 变量，集群客户端就被创建了。

```

// 创建用于通信的集群客户端
// 通过集群中的接待员对象提供服务
val clusterClient =
  system.actorOf(ClusterClient.props(initialContacts))

```

在不指定节点的情况下，集群客户端能够通过路径向特定的服务发送消息。集群中的接待员对象就像现实中的接待员一样。当客户端发送消息时，每个节点中的接待员对象会查明指定节点中是否含有接收该消息的对象。例如，如果你想使集群中的某个节点在 60 秒内不接收请求，而且不指定具体的节点，可以使用下面的代码做到这一点：

```

clusterClient ! ClusterClient.Send(
  "/user/requestLimiter",
  LimitForSeconds(60))

```

注意，这条消息是使用一致性散列（consistent hashing）算法通过随机方式向节点发送的，目的是在集群中的节点之间均衡客户端的连接负载。如果客户端想要向这个 Actor 对象在每个节点中的所有实例都发送一条消息，那么它就可以通过使用 `SendToAll()` 方法广播这条消息。在本例中，这样做会使所有节点在 60 秒内拒绝接收请求：

```

clusterClient ! ClusterClient.SendToAll(
  "/user/requestLimiter",
  LimitForSeconds(60))

```

集群客户端还拥有另一种工作模式，在这种工作模式中，服务不是按照已命名路径注册的，而是根据主题注册的。在这种情况下，客户端不会通过路径发送消息，而会将约定的字符串用作主题。所有订阅了接待员消息的 Actor 对象，都

会收到客户端根据这个主题发送的消息，要在接待员对象中注册集群中提供服务的 Actor 对象，可使用 `registerSubscriber()` 方法。

```
ClusterReceptionistExtension(system)
    .registerSubscriber(
        "requestLimit",
        requestLimiter)
```

一旦订阅了接待员消息的 Actor 对象被接待员对象注册了，那么集群客户端就可以使用 `Publish()` 方法向所有订阅者发布消息。在下面的例子中，所有使用 `requestLimit` 主题注册的 Actor 对象都会被发送 `LimitForSeconds` 消息。

```
clusterClient ! ClusterClient.Publish(
    "requestLimit",
    LimitForSeconds(60))
```

集群路由器 Akka 框架中的路由器以两种形式存在：分组和池。当以分组形式存在时，路由器会与一组已经存在的、在集群的多个节点中正在运行的 Actor 对象通信。当以池形式存在时，路由器能够控制 Actor 对象的生命周期，并且能够在不同节点中创建新 Actor 对象。下面详细介绍分组和池形式。

乍看之下，集群路由器分组好像完全没用。在这种形式中，路由器无法控制它为之提供路由服务的 Actor 对象的生命周期。因此，当路由器能够向负载最轻的节点发送消息时，它无法直接增加更多的 Actor 对象或节点。

然而，当你知道有多少个路由器为一个 Actor 对象分组提供路由服务时，分组形式就会变得更为有用。在配置集群角色时，这意味着集群中的许多节点可以被划分为做相同工作（通常为后台服务）的 Actor 对象分组。在这类情况中，许多前台处理程序都会拥有指向提供后台服务的 Actor 对象分组的集群路由器。分组路由器天生会通过多对多模式为 Actor 对象分组提供路由服务，如图 2.11 所示。

使用配置文件 `application.conf` 可以配置分组集群路由器。定义集群路由器，与定义普通的路由器很相似。其中的主要差异是，路由器是在 `cluster` 部分中定义的。

```
akka.actor.deployment {
  /groupRouter {
    router = consistent-hashing-group
    nr-of-instances = 100
    routees.paths = ["/user/backendService"]
    cluster {
```

```

    enabled = on
    allow-local-routees = on
  }
}
}

```

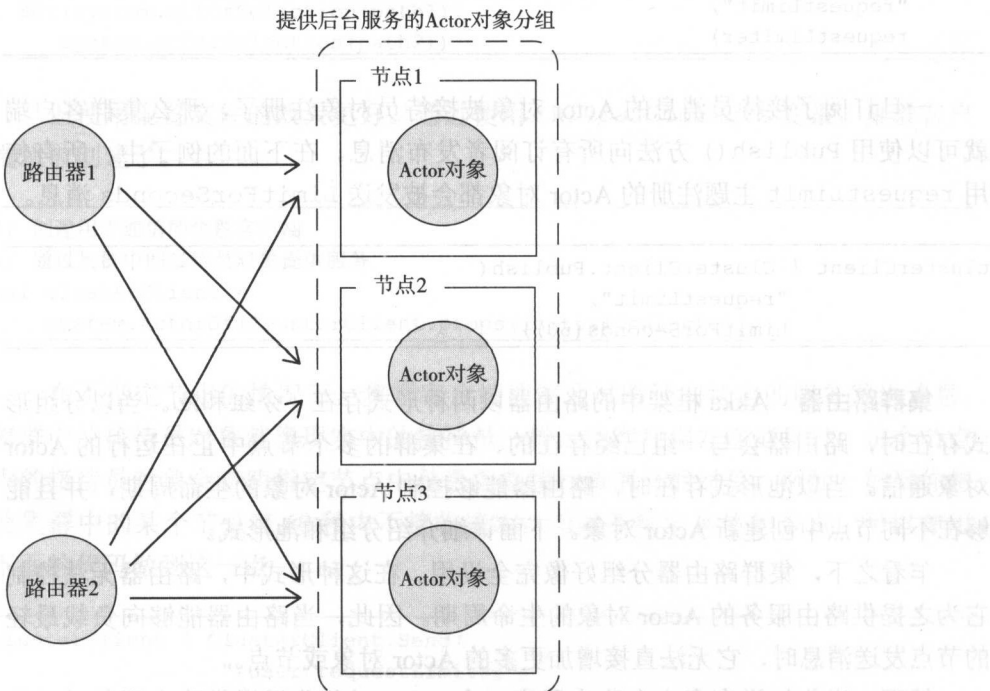


图 2.11 多个集群路由器为提供后台服务的 Actor 对象分组转发消息。

本例中的路由器是使用 `consistent-hashing-group` 定义的。这是一种很有用的集群路由器默认定义，因为这会以散列方式将节点排列成一个圆，然后向各个节点分配间隔（圆的组成部分），从而均衡集群中各个节点的负载，即使在向集群中添加节点和从集群中删除节点的情况中也能够做到这一点。其他类型的路由器（如 `random-group`）可根据具体情况选择使用。

在这个例子中，向外部提供服务的 Actor 对象的路径被定义为 `/user/backendService`（请参阅配置文件中的 `routees.paths` 部分），该 Actor 对象必须独立于路由器启动：

```

val backendService =
  system.actorOf(
    Props[BackendService],
    name = "backendService")

```

Actor 对象 `backendService` 启动后, 就可以启动分组路由器, 从而可以向后台服务发送消息了。注意, 如果通过路由器发送消息, 那么应使用 `ConsistentHashableEnvelope` (请参阅第 8 章) 封装器封装消息。该封装操作作为路由器的散列函数提供了关键信息, 使之能够均匀地在 Actor 对象之间分发消息。默认情况下, Akka 框架使用 `MurmurHash` 函数, 但也允许你使用自己定义的散列函数。

`ConsistentHashableEnvelope` 封装器听起来可能有点吓人, 但它唯一必做的工作是确保消息拥有合适的逻辑 ID (标识符)。你不必使用 `UUID` (全局唯一标识符) 或 `SHA-1` 之类的加密散列算法处理消息。如果你要发送 `QueryForUser(userId)` 消息, 可使用下面的方式设置关键信息:

```
val key = s"QueryForUser-${userId}-${System.currentTimeMillis}"
```

这样就可创建分组路由器并通过它发送消息了。请参阅前面的 `/groupRouter` 配置, 此处使用了与之相同的 Actor 对象名称。

```
val groupRouter =  
    system.actorOf(  
        FromConfig.props(Props.empty),  
        name = "groupRouter")  
groupRouter ! ConsistentHashableEnvelope(  
    message,  
    hashKey = message.id)
```

一旦该路由器和集群启动了, 向该路由器发送的消息就会被转发给集群中的 Actor 对象。随着提供服务的 Actor 对象增多, 它们会根据 `nr-of-instances` 值集, 被添加到路由器对象中。如果某个节点出了故障, 或者出现了其他不可用情况, 其中的 Actor 对象会自动被路由器取消注册资格。如果该服务中仅有一个实例被请求, 那么就可以将 `nr-of-instances` 变量设置为 1, 使分组路由器为集群内部请求该服务的节点提供代理服务。对于集群外部的 Actor 对象来说, 使用集群客户端更为合适。

下面将介绍池路由器。管理 Actor 对象池的集群路由器与管理 Actor 对象分组的集群路由器不同。在池形式中, 路由器负责管理它为之提供路由服务的 Actor 对象的生命周期, 如图 2.12 所示。出现这种情况的原因, 是这天生就是一对多的关系, 这意味着整个集群中仅存在一个为 Actor 对象提供路由服务的路由器。因此, 应该将路由器定义为集群单例对象。

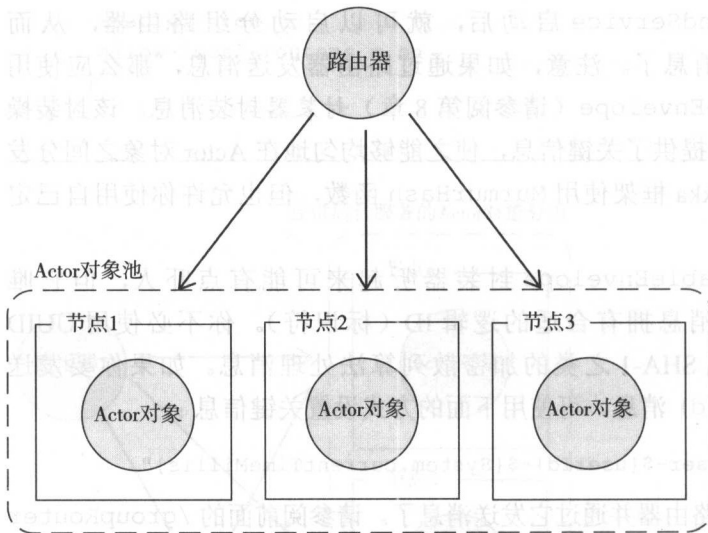


图 2.12 集群路由器负责管理 Actor 对象池。

在 Actor 对象扮演完资源密集型任务的工人角色时（通常会在专门分配的硬件中工作），池形式最有用。下面举例说明，一个集群接收音频文件（都是 50MB 无压缩 WAV 文件）并将这些文件转换为各种格式。

在这种情况下，定义管理 Actor 对象池的路由器可以取得最理想的效果。该路由器会接收 TranscodeJob 消息，告知 Actor 对象应将音频文件转换为哪种格式，并将转换任务分发给集群中不同节点中的不同 Actor 对象。许多高端计算机会被视为节点并被分配任务。每台用于完成转码任务的计算机都含有两块支持超线程技术⁸的 CPU，每块 CPU 中含有 4 个核心。

```
akka.actor.deployment {
  /singleton/transcodingService/poolRouter {
    router = consistent-hashing-pool
    nr-of-instances = 100
    cluster {
      enabled = on
      max-nr-of-instances-per-node = 16
      allow-local-routees = off
      use-role = transcode
    }
  }
}
```

⁸ 每个支持超线程技术的CPU核心都能够同时执行两个线程。因此，这种硬件架构可同时执行16个线程。

本例中的路由器是使用 `consistent-hashing-pool` 类定义的，并将 `max-nr-of-instances-per-node` 变量设置为 16，这会使每个支持超线程技术的 CPU 核心负责处理两个 Actor 对象。还应注意 `allow-local-routees` 变量被设置为 `off`，这些转码工作应在远程计算机上完成。

在转换文件格式时，还需要做一些额外工作。应命令 `TranscodeJob` 对象将具有唯一性的 ID 用作一致性散列函数的关键信息，这能够避免在向路由器发送消息时使用 `ConsistentHashableEnvelope` 封装器的麻烦。

```
import akka.routing.ConsistentHashingRouter._
ConsistentHashable
...
case class TranscodeJob(
  id: String,
  transcodeOptions: TranscodeOptions)
extends ConsistentHashable {
  override def consistentHashKey = id.toString
}
```

应定义一个 `TranscodingService` 类，将该类用作工人和路由器之间的中间人，确保复制工作不会被漏发，将执行失败的工作记录下来并重新执行。可在路由器的内部定义 `TranscodingService` 类。

```
class TranscodingService extends Actor {
  // 配置了 Actor 对象池的路由器
  val poolRouter =
    context.actorOf(
      FromConfig.props(Props[TranscodeWorker]),
      name = "poolRouter")

  def receive = {
    case job: TranscodeJob =>
      // 还没有接到工作
      // 重启执行失败的工作等
      val originalSender = sender()
      poolRouter ! job
      // 为工作设置时限
      case result: TranscodeResult =>
        // 用于处理结果的额外逻辑
        // 来自工人的消息等
  }

  object TranscodingService {
```

```
def props = Props[TranscodingService]
}
```

这样就在提供代理服务的路由器内部的每个节点中，创建了集群单例对象管理器。这个单例对象管理器只能管理带有 `transcode` 标记的节点。

```
system.actorOf(
  ClusterSingletonManager.props(
    singletonProps = TranscodingService.props,
    singletonName = "transcodingService",
    terminationMessage = PoisonPill,
    role = Some("transcode")),
  name = "singleton")
```

要向 `TranscodingService` 对象发送消息，应创建一个单例对象代理并向该代理发送消息。

```
val path = "/user/singleton/transcodingService"
val transcodingService =
  system.actorOf(
    ClusterSingletonProxy.props(
      singletonPath = path,
      role = Some("transcode")),
    name = "transcodingServiceProxy")
val transcodeJob = createTranscodeJob()
transcodingService ! transcodeJob
```

在使用池路由器时，所有 `Actor` 对象都会收到相同数量的转码任务。如果某个节点出了故障，该路由器会以平均方式将工作分发给 `Actor` 对象池中的可用 `Actor` 对象。如果集群中添加了新节点，那么该路由器就会自动在 `Actor` 对象池中启动新的 `Actor` 对象并开始向新的 `Actor` 对象发送工作。

根据负载扩展和收缩 前面介绍过，通过集群功能可以使 `Actor` 对象以透明方式跨节点相互通信，即使在通过动态方式向集群添加节点和从集群中移除节点的情况下，也可以做到这一点。前面还介绍了使用集群路由器通过一致性散列算法，均匀地向多个节点分发消息的方式。

一致性工作是指每条消息会占用大致相等的资源。如果类似的消息会生成差异较大的处理需求，那么通过平均分发消息的方式就无法解决这种问题。但路由器必须确保以平均方式向集群中的节点分配负载。

在前面的转码示例中，所有转码工作都是相同的，并且会占用相同数量的资源。然而，如果输入参数（音频文件的大小、原格式和转码类型）改变了，那么

即使某个倒霉的节点收到的消息数量与其他节点收到的消息数量相同，该节点可能会被分配给较困难的转码工作。

负载均衡路由器解决了这个问题。这种处理方式与一致性散列路由器处理 Actor 对象池和分组的方式类似，但会使用负载均衡路由器处理 Actor 对象池和分组。这些路由器会从每个节点收集信息，并根据这些信息分配工作，因此担负繁重工作的节点，不太可能会被再次分配工作。用于查明负载情况的信息有许多种。

- heap：用于查明 JVM 有多少可用堆内存。
- load：用于查明 UNIX 系统的平均负载（以 CPU 运行队列长度为基础）。
- cpu：用于查明 CPU 的使用情况。
- mix：用于查明堆内存、CPU 和负载的情况。

默认情况下，JMX 会收集这些信息，但是使用本地操作系统监视，不会获得太精确的结果。在情况允许时，应使用 Akka 集群功能中的 Hyperic Sigar 工具集，要在正式开发的产品中获得精确信息，我强烈建议你使用 Hyperic Sigar 工具集。Hyperic Sigar 是一个自带的软件库，要使用这个软件包，可在你项目的 build.sbt 文件中添加下面的代码：

```
libraryDependencies += "org.fusesource" % "sigar" % "1.6.4" %  
classifier("native") classifier("")
```

将集群路由器从一致性散列型转换为自适应负载均衡型比较简单。

```
akka.actor.deployment {  
  /singleton/transcodingService/poolRouter {  
    router = adaptive-pool  
    metrics-selector = load  
    nr-of-instances = 100  
    cluster {  
      enabled = on  
      max-nr-of-instances-per-node = 16  
      allow-local-routees = off  
      use-role = transcode  
    }  
  }  
}
```

此处唯一需要做的改动是，将 router 选项设置为 adaptive-pool，并将 metrics-selector 选项设置为 load。选择哪种负载度量标准选取器需要技巧，因为转码主要是一种仅与 CPU 能力有关的处理工作，所以我们的第一个感觉是查看 CPU 的使用率。然而，在 Akka 集群中被高效使用的计算机的 CPU 使用率都

会接近 100%，因此根据 CPU 运行队列长度（metrics-selector=load）观察负载情况可以取得更好的效果，因为这可以查明有多少进程正在等待使用 CPU。

下面介绍度量标准事件。负载均衡路由器使用的度量负载的信息可以被直接使用。要监听度量负载的事件，可以使 Actor 对象订阅 ClusterMetricsChanged 实例。将负载监控器创建为集群单例对象会很有用，因为它能够通过消息实时了解负载情况，比查询 JMX 中的托管 Bean 或使用 UNIX 命令行实用程序效果更好。

下面展示了一个负载监听器类：

```
import akka.actor.{Actor, ActorLogging, Address}
import akka.cluster._
import akka.cluster.ClusterEvent._
import akka.cluster.StandardMetrics.{Cpu, HeapMemory}
import akka.cluster.routing._

class CentralMetricsListener
  extends Actor with ActorLogging {
  val cluster = Cluster(context.system)
  var nodes = Set.empty[Address]
  var nodeCpu = Map[Address, Cpu]()
  var nodeHeap = Map[Address, HeapMemory]()
  var cpuCapacity = Map[Address, Double]()
  var heapCapacity = Map[Address, Double]()
  var loadCapacity = Map[Address, Double]()
  var mixCapacity = Map[Address, Double]()

  override def preStart(): Unit = {
    cluster.subscribe(self, classOf[MemberEvent])
    cluster.subscribe(self, classOf[ClusterMetricsChanged])
  }

  override def postStop(): Unit =
    cluster.unsubscribe(self)

  def receive = {
    case ClusterMetricsChanged(clusterMetrics) =>
      clusterMetrics.foreach { nodeMetrics =>
        val address = nodeMetrics.address
        extractHeap(nodeMetrics).map {
          heap => nodeHeap = nodeHeap + (address -> heap)
        }
        extractCpu(nodeMetrics).map {
          cpu => nodeCpu = nodeCpu + (address -> cpu)
        }
      }
    extractCapacity(clusterMetrics)
    logMetrics()
  }
}
```

```

case state: CurrentClusterState =>
  nodes = state.members.collect {
    case m if m.status == MemberStatus.Up => m.address
  }
  filterMetrics()

case MemberUp(member) =>
  nodes += member.address

case MemberRemoved(member, _) =>
  nodes -= member.address
  filterMetrics()

case _: MemberEvent => // ignore
}

def filterMetrics(): Unit = {
  nodeHeap = nodeHeap.filterKeys(
    address => nodes.contains(address))
  nodeCpu = nodeCpu.filterKeys(
    address => nodes.contains(address))
}

def extractHeap(
  nodeMetrics: NodeMetrics):
Option[HeapMemory] =
nodeMetrics match {
  case HeapMemory(address, timestamp, used,
    committed, max) =>
    Some(HeapMemory(address, timestamp, used,
      committed, max))

  case _ =>
    None
}

def extractCpu(nodeMetrics: NodeMetrics): Option[Cpu] =
nodeMetrics match {
  case Cpu(address, timestamp,
    systemLoadAverage, cpuCombined,
    processors) =>
    Some(Cpu(address, timestamp,
      systemLoadAverage, cpuCombined,
      processors))

  case _ =>
    None
}

```

```

def extractCapacity(
  nodeMetricsSet: Set[NodeMetrics]):
Unit = {
  loadCapacity =
    SystemLoadAverageMetricsSelector.capacity(
      nodeMetricsSet)
  heapCapacity = HeapMetricsSelector.capacity(
    nodeMetricsSet)
  cpuCapacity = CpuMetricsSelector.capacity(
    nodeMetricsSet)
  mixCapacity = MixMetricsSelector.capacity(
    nodeMetricsSet)
}

def logMetrics() = {
  nodes.foreach { node =>
    val heap = nodeHeap(node)
    val cpu = nodeCpu(node)
    log.info(s"port = ${node.port}, committed = "${
    ${heap.committed}, max = ${heap.max}, used = ${heap.used}")
    log.info(s"port = ${node.port}, cpuCombined = "${
    ${cpu.cpuCombined}, systemLoadAverage = "${
    ${cpu.systemLoadAverage}, processors = ${cpu.processors}")
    log.info(s"port = ${node.port}, loadCapacity = "${
    ${loadCapacity(node)}")
    log.info(s"port = ${node.port}, heapCapacity = "${
    ${heapCapacity(node)}")
    log.info(s"port = ${node.port}, cpuCapacity = "${
    ${cpuCapacity(node)}")
    log.info(s"port = ${node.port}, mixCapacity = "${
    ${mixCapacity(node)}")
  }
}
}

```

在使用 4 核 CPU 的 Macbook Pro 笔记本时, 在集群中运行的 CentralMetrics-Listener 对象会为单个节点记录下列日志:

```

port = Some(52038), committed = 487587840, max =
Some(954728448), used = 317080792
port = Some(52038), cpuCombined =
Some(0.9891756869275603), systemLoadAverage =
Some(3.84326171875), processors = 4
port = Some(52038), loadCapacity = 0.0391845703125

```

```
port = Some(52038), heapCapacity = 0.6678837918109255  
port = Some(52038), cpuCapacity = 0.010824313072439695  
port = Some(52038), mixCapacity = 0.23929755839862174
```

这段日志展示了每个负载度量标准选取器都返回了使用双精度数值代表的负载能力，1 代表完全空载，0 代表该节点正满负荷运行。这段日志还展示了这个节点拥有大量的可用 JVM 堆内存 (0.66)，而它的 CPU 负载能力 (0.01) 和整个节点的负载能力 (0.03) 几乎耗尽。最后，选取器 mix 将负载信息 load、mix 和 heap 汇总到一起，获得的结果为 0.23，这种查看负载情况的方式更好，但是无法查明系统瓶颈在哪里。

因为负载监听器能够监视所有节点并将相关的负载情况记录下来，因此可以更灵活地使用它，而不应仅将之用作调整整个集群行为的简单路由器。

当你发现整个集群的负载过高，而且没有多余的负载能力时，有两个选择：通过拒绝客户端的请求减轻应用程序的负载，或者提高集群的负载能力。

拒绝请求总会令人不快，因为这等于告诉用户，他们得不到他们想要的服务。下面列出了几种减少客户端请求数量的方式（按造成后果的严重程度排序）：

- 为客户端提供一种服务通道，期待它们能够相互礼让。
- 发送明确的消息，请客户端减少请求数量。
- 检查速度瓶颈或限制请求数量，并拒绝占用资源最多的客户端。
- 彻底拒绝一部分请求，尤其是低优先权的请求（如反复轮询数据的请求）。

此外，可以使用 Backpressure 模式通过内部有界队列管理工作，进而以自动方式管理负载。工作会由多个工人从队列中取出，而且当队列中的工作过多时，工作会被退回客户端。当队列中的工作量到达最高限制时，接收到的请求（向队列中添加工作的）会自动被拒绝，如图 2.13 所示。注意，这种内部队列与 Actor 对象的消息缓存不同，Actor 对象的消息缓存是不对 Actor 对象开放的，而队列应在 Actor 对象中明确定义。⁹

⁹ 第3章介绍了一种与此类似的工作队列，这种队列用于处理线程。

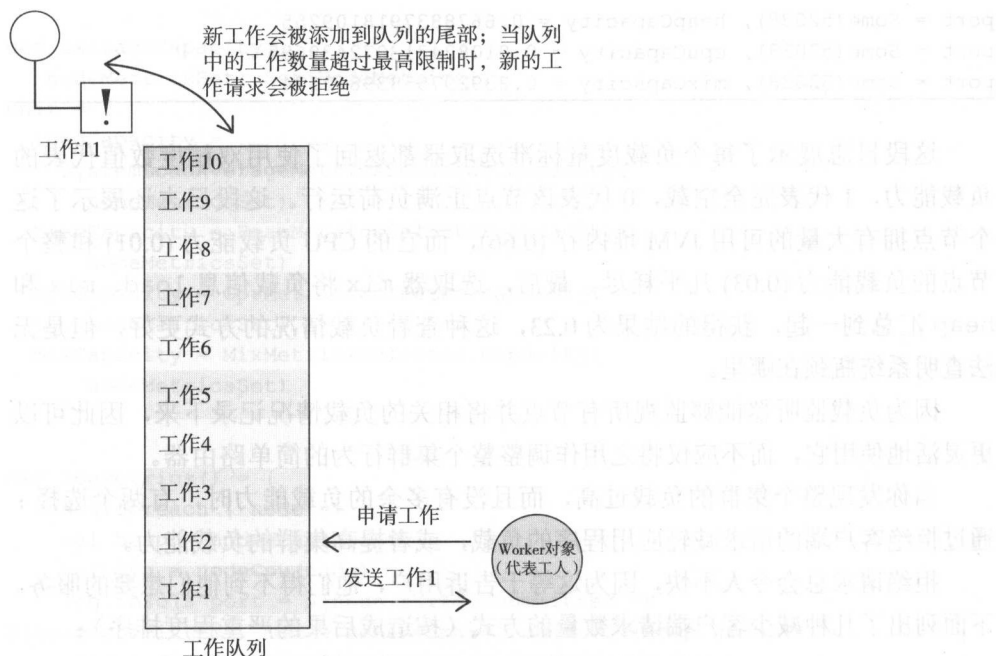


图 2.13 通过设置工作队列并为该队列设置最高容量或阈值（超过最高限制的新工作请求会被拒绝），可以应用回压模式。

第二个选项要好得多：当在整个集群中检测到高负载情况时，向资源供应者（如 Apache Mesos 集群管理器）发送一条消息，要求获得更多资源。当负载降低时，负载监听器可以向部分节点发送消息，命令它们关闭自己，以避免浪费资源。这种方式的处理细节超出了本书涵盖的范围，但我在撰写本书时这种处理方式正迅猛崛起为新的潮流。

测试 Actor 对象

阅读了前面介绍 Actor 模型的内容后，你可能已经考虑到使用怎样的方式测试 Actor 对象。可能你已经料到这是一件困难的任务，因为 Actor 对象的外部会发生许多事情，Actor 对象内部应该拥有怎样的灵活度才能对这些事情做出回应呢？

因为 Actor 对象的外部使用异步处理方式，而且 Actor 对象通常无法以确切的、预先决定好的、有序方式做出回应，所以 Actor 对象无法确定收到消息的顺序是否正确。这就是第 1 章介绍过的非确定性。然而请注意，可以将用于保存和管理状态的 Actor 对象设计为有限状态机。这使你能够通过设计，使 Actor 对象了解指定的状态、在该状态中可以处理哪类消息、通过怎样的方式切换当前状态和收

到当前状态无法处理的消息时应该怎样做。然而，这属于设计范畴而不是测试范畴。

令人高兴的是，Akka 框架开发团队了解这一情况，因为他们自己也必须测试 Actor 对象，以确保 Akka 框架的健壮性。当你必须为自己做饭时，你通常会努力将饭做得美味；你的目标是尽量使烹饪工作简单，尽量使饭菜美味。同理，你也可以期待 Akka TestKit 测试包具有既简单又好用的特质。

Akka 框架开发团队提供了两种测试 Actor 对象的方式。第一种方式用于处理 Actor 状态机的适应度。第二种方式用于处理 Actor 对象通过处理消息对外界做出回应的方式。

Actor 对象单元测试

如果你熟悉 Java 或 C# 中的单元测试方式，就会对 Akka 框架中的单元测试感到熟悉并能够轻松掌握它。在开始测试前，必须先导入 TestKit 测试包，创建 Actor 对象，并获得该 Actor 对象的基于测试的 ActorRef，并获取基础 Actor 对象的引用。

```
import akka.testkit.TestActorRef

val riskManagerRef = TestActorRef[RiskManager]
val riskManager = actorRef.underlyingActor
```

不要尝试直接实例化 Actor 对象，这不会有任何好处。即使能够自由访问 Actor 对象的方法，但 Akka 框架仍旧会阻止你直接使用 Actor 对象。如果你尝试创建新的 Actor 实例，就会遇到下面这类运行时错误：

```
Exception in thread "main" akka.actor.ActorInitializationException: You cannot create an
instance of [ActorClassname] explicitly using the
constructor (new).
```

TestActorRef 引用使你能够避开常规限制，甚至能够使你获得 Actor 实例的引用。除非只能通过更高级的操作调用 Actor 对象内部的方法，否则为了获得可测试性就不应将 Actor 内部的方法设置为私有方法。不存在隐藏高级操作的必要，因为在正常的运行环境中这些操作无法被直接调用。换言之，在正常情况下，发送消息是外部事物使 Actor 对象工作的唯一方式。

TestActorRef 引用的另一个优点是，通过它发送消息可以将消息直接发送给 receive 偏函数，而不是通过异步方式将消息送入 Actor 对象的消息缓存中。

只要 Actor 对象参与了工作，那么它同一时刻就只能处理一条消息，因此无须对 Actor 对象的可测试性做额外的处理。

掌握了这些知识后，你就可以使用任何常见的基于 Scala 或 Java 的测试框架。例如，你可以使用 ScalaTest 测试框架 [ScalaTest]。在这种情况下中应在项目的 build.sbt 文件中添加下面的代码：

```
libraryDependencies += "org.scalatest" % "scalatest_2.11" %
  % "2.2.0" % "test"
```

如果你使用 Maven 项目管理工具，则应在该文件中添加下面的代码：

```
<dependency>
  <groupId>org.scalatest</groupId>
  <artifactId>scalatest_2.10</artifactId>
  <version>2.2.0</version>
  <scope>test</scope>
</dependency>
```

如果你使用 Gradle 自动化构建工具，则应在该文件中添加下列代码：

```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath 'com.github.maiflai:gradle-scalatest:0.4'
  }
}

apply plugin: 'scalatest'

dependencies {
  compile 'org.scala-lang:scala-library:2.10.1'
  testCompile 'org.scalatest:scalatest_2.10:2.0'
  testRuntime 'org.pegdown:pegdown:1.1.0'
}
```

下面让我们使用 ScalaTest 测试框架和其中的 FunSuite 工具更轻松地来进行 JUnit 测试：

```
import org.scalatest.FunSuite
import akka.testkit.TestActorRef
```

```
class RiskManagerTestSuite extends FunSuite {
  val riskManagerRef = TestActorRef[RiskManager]
  val riskManager = actorRef.underlyingActor

  test("Must have risks when told to calculate.") {
    riskManagerRef ! CalculateRisk(Risk(...))
    assert(!riskManager.risksToCalculate.isEmpty)
  }
  ...
}
```

应仔细观察 `ScalaTest[ScalaTest]` 测试框架中所有的优秀测试功能，其中包括 `xUnit`、`Behavior-Driven Development` 和一些介于这两种工具之间的功能。这样你会发现必须模拟一些组件（如 `ActorRef` 实例），以便获取或指向 `Actor` 对象。

行为型测试消息处理

行为驱动开发（`Behavior-Driven Development`, `BDD`）和实例化需求（`Specification By Example`, `SBE`）是指一种将重点放在业务需求上的测试模式。这类测试的目的是反映出真正的业务场景，并将这些场景用作软件的需求。在使用 `BDD/SBE` 时，你的目标是使所有相关客户认同你归纳出的需求，并将这些需求编写为一系列标准测试文档。这与创建多种需求文档、进行一次性测试或反复测试以及典型项目中的软件模型的思想背道而驰。这样做的目的是通过基于需求的测试中的断言验证行为，从而精雕细琢地创建软件模型。

`Akka TestKit` 测试包提供了一个名为 `TestKit` 的类，使用该类可以进行 `BDD` 测试。¹⁰ 要进行稳定性测试，你还必须混入许多 `Scala` 类或特征。下面是一个需求/测试规范的例子：

```
import org.scalatest.BeforeAndAfterAll
import org.scalatest.WordSpecLike
import org.scalatest.matchers.Matchers
import akka.testkit.ImplicitSender
import akka.testkit.TestKit

class RiskCalculatorSpec(testSystem: ActorSystem)
  extends TestKit(testSystem)
  with ImplicitSender
```

¹⁰ `Akka` 框架开发团队将这种测试称为整合测试，因为该测试的目的是检验将你编写的 `Actor` 对象与它周围的许多正在系统中运行的 `Actor` 对象整合到一起后的效果。其他人会将这种测试称为 `BDD/SBE` 测试。

```

    with WordSpecLike
    with Matchers
    with BeforeAndAfterAll {

// 此处存在一个隐含的类参数
def this() = this(ActorSystem("RiskCalculatorSpec"))

    override def afterAll {
        TestKit.shutdownActorSystem(system)
    }

    "A RiskCalculator" must {
        "confirm that a risk calculation request is in progress" in {
            val riskCalculator = system.actorOf(Props[RiskCalculator])
            riskCalculator ! CalculateRisk(Risk(...))
            expectMsg(CalculatingRisk(...))
        }
    }
}

```

这只是一个基于需求的测试和使用 TestKit 测试包的简单例子。你可以参阅 Ray Rostenburg [Roestenburg] 网站提供的示例，该网站详细介绍了 Akka TestKit 测试包提供的一系列基于需求的测试工具。

CompletableApp 类

在下面将要介绍的代码示例中，你看到的第一个神秘对象将会是 CompletableApp。CompletableApp 类扩展了 Scala 标准的 App 特征，而且增加了一点额外行为。为简便起见，它创建了一个 ActorSystem 实例。扩展该类的 Scala 对象（如 RequestReplyDriver），会拥有完成步骤计数功能。当你看到某个 Int 型值被赋予 CompletableApp 构造器时，该值设置的就是整个处理过程将要执行的可完成步骤数量。一旦所有步骤都完成了，CompletableApp 对象就会停止并关闭 Actor 系统。这可以使程序的主线程一直等待，直到各种 Actor 对象都完成了预定的所有步骤为止。

不要在正式的软件产品中使用 CompletableApp 类

等待许多步骤被完成或一直等待，与编写高吞吐量的响应式应用程序的目的不符。等待主测试系统对象 CompletableApp 完成处理步骤，仅是一种易于查明代码执行情况的方式。因此，不应在正式的软件产品中使用它。

向能够以适当方式输出结果的 Actor 对象发送事件消息（请参阅第 6 章），是

好得多的获取完成步骤计数的方式。那么应该向哪个 Actor 对象发送事件消息呢？应通过向完成处理步骤的 Actor 对象发送消息，将之介绍给主处理过程中的 Actor 对象。可以通过处理过程管理器（请参阅第 7 章）、消息总线（请参阅第 5 章）与管道和过滤器链条（请参阅第 4 章），控制基于消息的介绍操作。例如，当处理过程完成了最后一个处理步骤后，会由处理过程管理器本身向已知的完成处理步骤的 Actor 对象发送事件消息。

下面是 CompletableApp 类的源代码：

```
package co.vaughnvernon.reactiveenterprise

import akka.actor._

class CompletableApp(val steps:Int) extends App {
  val canComplete =
    new java.util.concurrent.CountDownLatch(1);
  val canStart =
    new java.util.concurrent.CountDownLatch(1);
  val completion =
    new java.util.concurrent.CountDownLatch(steps);

  val system = ActorSystem("ReactiveEnterprise")

  def awaitCanCompleteNow() = canComplete.await()

  def awaitCanStartNow() = canStart.await()

  def awaitCompletion() = {
    completion.await()
    system.shutdown()
  }

  def canCompleteNow() = canComplete.countDown()

  def canStartNow() = canStart.countDown()

  def completeAll() = {
    while (completion.getCount > 0) {
      completion.countDown()
    }
  }

  def completedStep() = completion.countDown()
}
```


在使用 `RequestReplyDriver` 对象的情况中，会在 `CompletableApp` 基础对象中配置许多步骤，但是其他 Actor 对象必须帮助驱动测试的对象（如 `RequestReplyDriver`）倒计时。下面列出需要特别处理的位置：

```
...  
RequestReplyDriver.completedStep()  
...
```

可以在特殊环境中使用 `CompletableApp` 对象中的其他方法。例如，在主处理过程开始前，为了设置许多 Actor 对象，必须使用 `awaitCanStartNow()` 方法暂停驱动测试的对象。一旦 Actor 对象完成了运行的准备工作，其中某个 Actor 对象会调用 `canStartNow()` 方法。

因此，这个标准 App 类的特例能够管理每个 Actor 对象处理步骤的最终完成情况。本书的代码示例会大量使用 `CompletableApp` 对象。但是，你应该将其从正式软件产品中去除，或者有保留地使用它。不应将 `CompletableApp` 类视为 Akka 开发中的标准组成部分，它仅是一种类似脚手架的辅助工具，使代码变得更易于阅读和理解。

小结

本章是一篇详细的 Scala 和 Akka 指导材料。下面是本章的要点：

- 你可以通过多种方式获得 Scala 语言和 Akka 框架，也可以根据自己的喜好直接下载这些工具。本章概括介绍了 Scala 编程语言，还介绍了命名软件包和导入 Scala 工具的方式，从而使你能够使用条件、循环和各种迭代结构。
- 本章详细介绍了使用 Akka 框架编写程序的方式。其中包括：创建 Actor 系统、创建 Actor 对象、向 Actor 对象发送消息和创建 Actor 对象的监督层次结构。
- 本章还介绍了一些高级内容，如远程处理和集群。
- 最后，本章详细介绍了测试 Actor 对象的基础知识。

下一章将介绍在开发追求高性能的应用程序时，Actor 对象能够起到的重要作用，以及一些并发处理工具。

第 3 章

性能情结

晶体管和时钟频率！这就是使我们这些计算机专业人员一直沉迷其中的两件事物。

在你编写程序时，是否会花许多精力考虑处理器的性能？大约半个世纪以来，晶体管和时钟频率就在软件系统中扮演着戏份以指数形式增加的角色。我们沉醉于加速增加晶体管的数量，沉醉于无限提高时钟频率。因为这可以使我们编写的越来越复杂的程序以越来越快的速度运行，我们热衷于使用更多的晶体管和吞吐量更大的集成电路。

早期的晶体管计算机是由那些努力将数千个晶体管放在一起工作的人们制造出来的。当前，处理器中集成了数百万、数千万个晶体管，这类处理器被安装在遍布世界的大大小的计算机中。在我们这类人给予晶体管巨大期望的情况下，晶体管好像也没有辜负我们的期望，一直在不断提高计算机的性能。但是请注意，未来的晶体管可能与我们过去享用过的晶体管性能盛宴大不相同。

毫无疑问，晶体管和时钟频率一直处于不断改变的状态中。

晶体管

1925 年 10 月 22 日，Julius Edgar Lilienfeld（一位奥匈帝国的物理学家）在加拿大申请了第一个晶体管专利。令人遗憾的是，没有多少人知道他的成就，晶体管从默默无闻到举世闻名花了 20 多年的时间。早在 1906 年，电子管就被发明出来用作电子放大器，它为人类迎来了电子时代。在 Julius Edgar Lilienfeld 的发明仍旧不为大多数人所知的情况下，在很长一段时间内电子管被用作电力电子设备（三极管）[Triode]。

直到 1947 年，科学家们才使用晶体管取得了成功，真正将晶体管应用到产品中又花了不少时间（1953 年）。第一个使用晶体管的实践案例出现在 1954 年。是在计算机中使用晶体管吗？对这个问题，既可以回答是也可以回答否。实际上，第一台晶体管计算机是在 1953 年由萨塞克斯大学的 Dick Grimsdale 发明的。然而，当时晶体管本身的不可靠性阻碍了他的计算机发明工作。晶体管计算机还必须等待

它命运的转机。在被广泛地用于商业用途前 [Transistor], 晶体管必须经过多次精炼。

同年, Dick Grimsdale 发明的晶体管计算机取得了有限的成功, 晶体管第一次被成功用作商业无线电材料。一台 TR-1 收音机总共用了 4 枚晶体管, 售价 49.95 美元。考虑到通货膨胀因素, 当时购买一台 TR-1 收音机, 大约相当于现在购买一台价值 449.95 美元的 iPod 播放器。尽管价格较贵, 但晶体管收音机比同时代的电子管收音机更优秀。晶体管体积更小、耗电量更少, 而且不必像电子管收音机那样花费过多时间进行预热。当时晶体管收音机确实是一件奢侈品, 晶体管收音机支持随开即用的功能 [Transistor]。

所有对晶体管的研究和开发导致大量晶体管收音机被生产出来 (TR-1 收音机大约售出了 150,000 台), 这一情况反过来又推动进一步对晶体管进行精炼, 以便在商业中更广泛地使用它, 这对于计算机的发展起到了最为重要的推动作用。

商业晶体管计算机的第一个成功, 归功于控制资料公司和西摩·克雷领导的团队, 该团队在 1960 年制造了巨型计算机 CDC 1604 [Cray-CDC]。但是, 显然第二代 IBM 7090 机 [IBM-History] (基于晶体管的) 是在 1959 年 11 月制造的, 它比 CDC 1604 早出现了几个月。第二代 IBM 7090 机比使用电子管处理器的第一代 IBM 7090 机快 6 倍。后来, 一台含有约 50,000 枚晶体管的 IBM 7090 机售价约为 290 万美元。在那个时代, 收音机不再是唯一的奢侈品。

经过这段不长时间的发展后, Intel 的创始人意识到晶体管对计算能力的提高会成为一种持续很长时间的潮流。1965 年, 戈登·摩尔发现将大量晶体管集成到电路中能够成倍提高速度, 因此他预言这种趋势会一直持续, 每两年计算机中的晶体管数量就会翻倍, 如图 3.1 所示。戈登·摩尔的这项发现被称为摩尔定律, 50 多年来这条定律一直是真理。实际上, 摩尔定律已经不再仅仅是一个预言或趋势。摩尔定律已经被证实是真理或者其中的某一部分是真理, 因为它已经成为半导体行业调整其发展步伐的业内标准 [Moore's Law]。

因此, 从 1959 到 2014 年, 半导体行业都基本上按照这个步调前进。但我们必须要问, 这种迅猛的发展趋势还会维持多久? 在 2005 年, 有人估计该趋势仅能持续到 2015 年或 2020 年。但 2010 年的半导体行业发展报告表明, 2013 年年末这个发展势头会变慢。据说过了这个时间点后, 晶体管的数量和密度只能以每三年翻倍的速度增长。戈登·摩尔在 2005 年发表声明, 说他提出的摩尔定律可能仅是一个预言, 无法永远反映晶体管的发展规律。戈登·摩尔指出随着时间的推移, 晶体管的体积会变得越来越小, 直到接近原子的体积, 这时晶体管的发展就会到达物理极限。

基于有可能出现的惊人技术突破 (如增加处理器的尺寸、量子计算和使用云计算完全取代集成电路的未来概念), 另一些人持比较乐观的观点。尽管戈登·摩

尔在 2005 年提出电子芯片中的晶体管数量最终不会超过数亿个，但当今的处理器都超过了这个晶体管数量预估极限。不仅是处理器，就连 Nvidia 公司生产的 GPU 芯片也能够含有 7.08 亿个晶体管。这个数值是 IBM 的大型机 zEC12 所用 CPU 中所含晶体管数量的 3 倍左右，zEC12 [IBM-zEC12] 使用的 CPU 是当今最快的 CPU 之一，其中含有 2.75 亿个晶体管 [Transistor Count]。

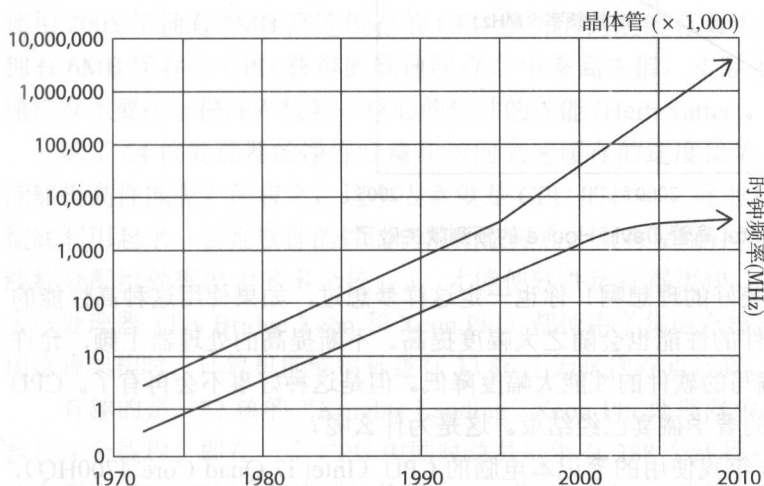


图 3.1 从 1973 年至 2010 年，根据摩尔定律预测的晶体管数量和时钟频率的增长情况。

虽然还不能确定，但 CPU 中晶体管数量以指数形式增长的迅猛势头似乎要走到尽头了。

时钟频率

计算机性能还由另一个因素决定。这个因素的重要性甚至超出了摩尔定律，Intel 的高管 David House 预测，芯片性能每 18 个月会翻倍。他之所以敢于做出这个超出摩尔定律的预测，是因为成倍增长的不仅仅是晶体管的数量，处理器的主频（时钟频率）也在不断提高。

当晶体管数量增长的速度即将走到 50 多年前有人预测的尽头时，处理器的时钟频率的提高速度已经一落千丈。Herb Sutter [Herb Sutter] 提出，尽管 David House 的预测对于将来数代处理器来说还是正确的，但是从 2003 年年初开始，CPU 时钟频率的提高速度就开始急剧下降，如图 3.2 所示。换言之，即使晶体管的数量还会根据摩尔定律继续增长，但在符合 David House 预言的情况下，CPU 主频的高增长速度只能再维持 10 多年（从 2003 年算起）。如果现实情况真是如此，那么在 2005 年时 CPU 的主频就会达到约 10GHz。

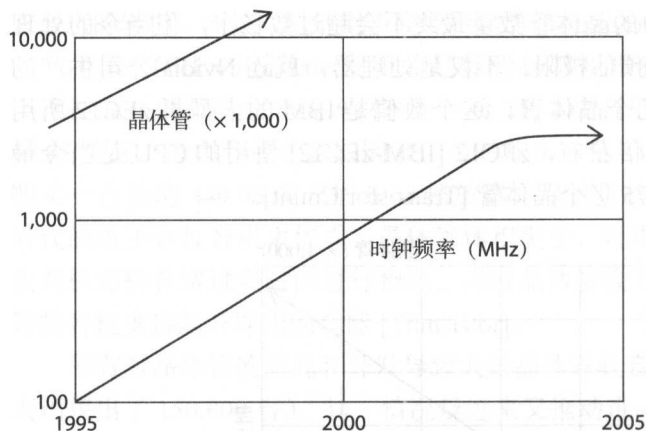


图 3.2 在 2003 年，Intel 高管 David House 的预测就失败了。

这是一种多么美好的理想啊！你也一定这样梦想过，如果使用这种高性能的硬件，你编写的软件的性能也会随之大幅度提高。不断提高的处理器主频，允许你每 18 个月将你编写的软件的性能大幅度降低。但是这种好事不会再有了。CPU 主频不断高速增长의 奢华盛宴已经结束。这是为什么呢？

即使到了 2014 年我使用的笔记本电脑的 CPU（Intel i7 Quad Core 4700HQ），仍旧在以 2.4GHz 的速度运行。比这块 CPU 快的 Intel 处理器有很多，但它们的主频也没有快太多（约 3.9GHz）。CPU 仍旧在发展，仍旧在不断增加其中的晶体管数量，但 CPU 的主频不再以翻倍的速度增长了。在 2003 年时，CPU 时钟频率的指数式增长势头就走到了尽头。下面是导致该情况出现的主要原因：

- 如果继续提高 CPU 的时钟频率，就无法通过传统方式为其散热。
- 如果继续提高时钟频率会导致 CPU 的功耗令人无法接受。

因此，怎样才能使当代的软件符合不断增长的性能需求呢？

核心和高速缓存

下面列出了与十年前的处理器相比，当前处理器的主要优势，现在这些特点即使是笔记本电脑也能够拥有：

- 64 位指令集（与 2005 年的 32 位处理器相比）。
- 同时能够处理 8 个线程的 4 核心 CPU（与 2005 年的可能还支持超线程技术的单核处理器相比）。
- 6MB 的高速缓存（与 2005 年的 2MB 高速缓存相比）。

从 32 位处理器转到 64 位处理器，并不总是能够自动获得性能提升。尽管 64 位 CPU 拥有更多寄存器、更高级的指令集和其他优化功能，但增加指针尺寸可能会导致负面效果。例如，如果你编写的软件大量使用了指针，那么 CPU 的高速缓存就会以更快的速度被填满。当 CPU 的高速缓存被填满后，处理器就不得不更频繁地以慢得多的速度与内存交换程序数据。因此，在执行你编写的代码时，使用 2005 年拥有 2MB 高速缓存的 CPU 获得的高速缓存命中率，会比使用当前拥有 6MB 缓存的 CPU 获得的高速缓存命中率高 3 倍。实际上，增加 CPU 的高速缓存主要用于提高不依赖多核心的软件的性能 [Herb Sutter]。

除了 64 位处理器的净吞吐量和访问高速缓存的速度优势外，还有哪个因素能够提高性能呢？简言之，这个因素就是 CPU 的核心。除非将增加的 CPU 核心彻底利用起来，否则软件的性能仍旧无法提高。只有使用并发编程技术（将多个线程分配给处理器中的多个核心），才能使软件运行得更快。Intel 的一系列顶级多核处理器（Ivy Bridge Xeon 和 Xeon Phi）都使用了集成众核架构（MIC），在使用这种架构时，计算机能够每秒进行 33.86 千万亿次浮点运算。

有趣的是，72 核的 “Knights Landing” Xeon Phi 处理器中的每个核心都能够执行 4 个线程。即在一个 CPU 中同时总共能够有 288 个线程被执行。与 1960 年的大型机 IBM 7090（每块芯片价值 4,000 美元）相比，顶级的 Xeon Phi 服务器廉价得就像二手货。当然，我们这些程序员必须设计好代码，才能让这 72 个核心同时运行 288 个线程。否则，恐怕会有 71¼ 个核心被闲置。这是为什么呢？注意，下面是一段 Intel Xeon Phi 服务器研究人员的重要报告。

研究人员对该型服务器进行了实验性能和可编程性研究。该服务器的制造商提出，要使用 Xeon Phi 服务器获得高性能，仍旧需要程序员提供帮助，仅使用传统编程模型中的编译器就获得高性能的目标仍旧很遥远。摘自 Fang Jianbin、Sips Henk、Zhang Lilun、Xu Chuanfu、Yonggang Che、Varbanescu AnaLucia 在 2014 年《ACM/SPEC 性能工程国际研讨会》提交的报告《Intel Xeon Phi 运行测试》。请浏览 http://en.wikipedia.org/wiki/Xeon_Phi。

现在，未来计算性能的重担不是仅由处理器担负了，程序员也必须帮助分担，这几乎重现了 1981 年注重软硬件兼容的形势。这就像我们成了 Intel 的集成电路工程师，我们的工作使下一代软件运行得更快，但我们要通过编程使软件运行得更快，而不是仅仅依靠硬件。

可伸缩性

20 世纪 90 年代后期, 面对软件性能挑战, 一种流行的解决方案是用性能更高的硬件砸倒这些问题。我曾经为一个项目提供咨询服务, 这个项目的很清楚: 基于 Java 的电子商务系统无法在现有的 Sun Enterprise 450 服务器上运行。怎么解决这个问题呢? 买一台 Sun Enterprise 10000 服务器。这款服务器的默认配置中使用了多块 UltraSPARC II 处理器, 而且能够根据需要进行添加更多处理器, 最高可达到 64 块, 这好像是一种理想的可伸缩途径。如果你不同意这个观点, 请咨询 Sun 公司的销售人员。

如果你曾经处理过大型 Java 服务器项目, 那么可能会记起 Java 2 平台企业版 (J2EE) 严格禁止你自己创建和管理线程。这个架构独占了线程的控制权, 你和你的应用程序都无法干涉。如果你不遵守这条禁令, 你创建的系统就会陷入混乱。当然, 随意创建线程会在 J2EE 容器或仅在 Java 虚拟机 (JVM) 导致潜在的问题。根据我的经验, 应用程序多使用一些进程不会导致灾难性问题。然而, 多线程技术还不是解决应用程序问题的通用方案。当时通过购买更多 Sun Enterprise 10000 服务器获得的效果, 可能比在容器中增加线程进而导致潜在问题所获得的效果更好。

几乎在同一时间段, 整个计算机行业内都流传着“扩展”消息, 几位创新者证实扩展规模是更为强力、实用和通用的选择。因特通 (Inktomi) 公司搜索引擎部门中的一些研究和开发人员就是一些先行者。如因特通公司的 Eric Brewer 博士所说, 他们的目标是通过大量的普通工作站节点, 创建一台超级计算机。他们为什么会设定这样的目标呢? 因特通公司正面临两大挑战。首先, 即使在处理器速度不断迅猛提高的时代, Web 文档和用户的增长速度已经超过了硬件性能的提高速度。其次, 因特通公司需要通过扩展规模, 使它的爬虫程序和搜索引擎满足信息时效性和用户的要求, 这两类需求都在全世界范围内迅速增长。否则, 因特通公司的解决方案只会使世界网络更加拥堵。即使在互联网泡沫没有破灭的时代, 在世界各地部署高端服务器需要的高投资也让人难以接受 [Brewer-Inktomi]。

因特通公司架构发展方向的一个关键推力是, 与需要等很长时间才能获得高端服务器相比, 现在该公司就能够购买并部署普通的硬件。因为通过增加普通硬件也能提高性能 (当时这个势头还在不断发展), 可以将较新的硬件部署到较旧、较慢的节点中。通过配置可以将更多的处理工作分配给较新的节点。集群工作站架构与高速网络结合到一起, 能够在不限制向集群中添加的工作站数量并不大幅度降低集群性能的情况下, 专门用于处理可伸缩型工作。此外, 该架构支持在不对并发集群操作产生负面效果的情况下, 以递增方式添加工作站 [Inktomi-Architecture]。

在搜索引擎领域中因特通公司的另一竞争者，使用类似的方式处理可伸缩性。Google 将普通的 x86 计算机用作服务器。近年来，Google 考虑在他们的服务器硬件中使用含两块双核处理器、大量的随机存取存储器（RAM，即俗称的内存）和两块 SATA 硬盘的自定义系统 [Google-Platform]。图 3.3 展示了 Google 的参考架构 [Google-Architecture]。

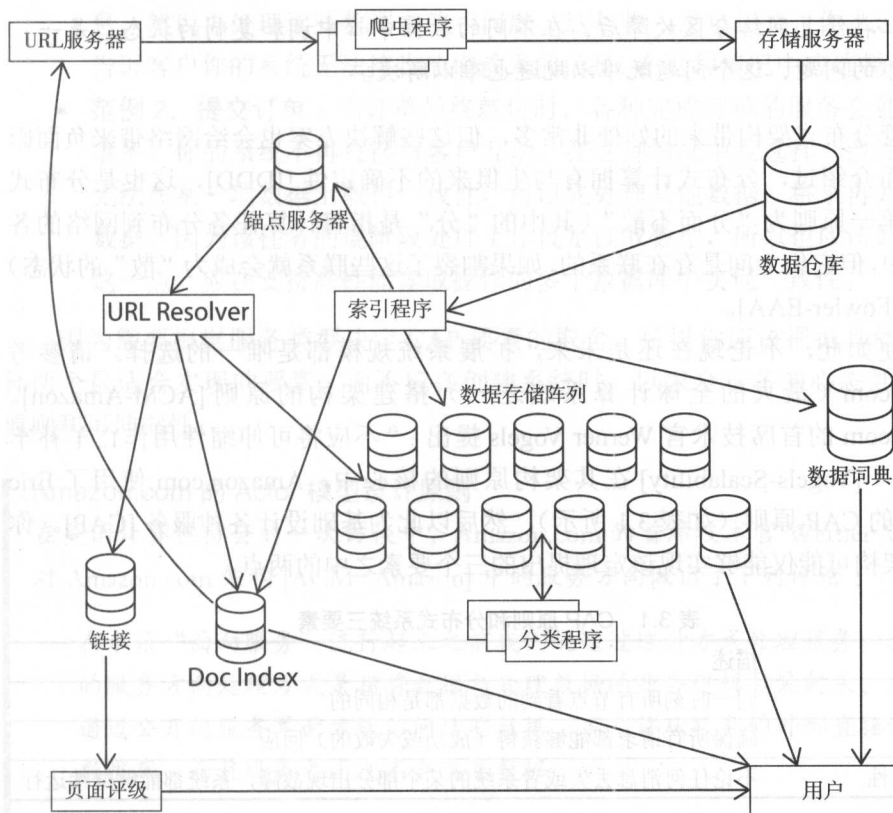


图 3.3 Google 在它的搜索引擎参考框架中，使用大量的普通 x86 计算机扩展集群规模。

像因特通公司一样，Google 的搜索引擎也需要使用快速的网络，支持图 3.3 所示的分布式操作。尽管其中的网络细节是顶级机密，但是根据 Google 透露的部分信息，我们可以推测出 Google 私网拓扑结构和性能的一点细节。我推测 Google 的私网使用专门定制的、高基数交换路由器，这类交换路由器中的每一台都拥有 128 个传输速率为 10 Gb/s 的以太网端口。为了进行冗余，其数据中心中至少会有两台这样的交换路由器。根据已经公开的信息，Google 私网能够以每秒 1Tb 的速度调整自身的服务规模。这真是一个速度非常快的网络 [Google-Platform]。

不论你的网络具有怎样的高速度（不太可能超过 Google 私网的速度），仍旧

会面对极为严峻的分布式计算的挑战。例如,至少 Google 已经在调查导致其网络运行中断和网络分区 [Google-Owies] 故障的一些原因¹。如果总是不能立刻修复网络分区故障,那么 Google 的软件系统和应用程序会怎么样呢?我们不了解具体的细节,但显然需要处理下列问题:

开发易于使用的抽象,解决因更新某一状态的多个版本引发的冲突问题。换言之,在修复网络分区故障后,在不同的数据中心中调和复制的状态,是一个困难的问题,这个问题既难以规避也难以解决。

尽管分布式架构带来的好处非常多,但这些解决方案也会给网络带来负面影响。前面介绍过,分布式计算拥有与生俱来的不确定性 [IDDD],这也是分布式计算的第一原则为“分而不散”(其中的“分”是指将计算任务分布到网络的各个部分中,但它们之间是存在联系的,如果割裂了这些联系就会成为“散”的状态)的原因 [Fowler-EAA]。

即便如此,不论现在还是未来,扩展系统规模都是唯一的选择。请参考 Amazon.com (最大的全球计算系统之一)搭建架构的原则 [ACM-Amazon]。Amazon.com 的首席技术官 Werner Vogels 提出:“不应将可伸缩性用作亡羊补牢的手段。” [Vogels-Scalability] 在其架构原则的核心中,Amazon.com 使用了 Eric Brewer² 的 CAP 原则 (如表 3.1 所示),然后以此为基础设计各种服务 [CAP]。你设计的架构可能仅能够实现该定理提出的三个要素之中的两点。

表 3.1 CAP 原则和分布式系统三要素

属性	描述
一致性	同一时刻所有节点看到的数据都是相同的
可用性	确保所有请求都能够获得 (成功或失败的) 回应
分区容忍性	不论任何消息丢失或者系统的某个部分出现故障,系统都能够继续运行

CAP 原则提出无法在一个系统中同时实现这三个要素,因此你必须在这三个系统属性中选择两个。Amazon.com 果然在其系统中实现了该定理中的第一个要素:分区是进行伸缩的必由之路,而 Amazon.com 一定要具有可伸缩性。因此,它就必须实现分区容忍性。当将分区引入网络时,就意味着必须处理各种原因导致的不一致性。然而,Amazon.com 还是选择拥有可伸缩性。失之东隅,收之桑榆是必然的结果。

了解了这些背景后,你还必须了解 Amazon.com 开发软件的方式。该公司的

¹ 网络分区故障是指导致网络被分割开的网络设备故障。

² 前面介绍过的因特通公司的系统架构,就是由 Eric Brewer 博士设计的。

开发团队都比较小，而且每个团队都负责一个单独的服务 [Vogels-Scalability]。这些服务有多大规模呢？一个页面中就会包含超过 100 个服务。根据服务的类型，Amazon.com 允许开发团队在一致性和可用性之间进行取舍。在选择这些要素时，可使用下列指导原则 [Vogels-AsyncArch]。

- **范例 1，购物车输入：**永远都应该先接收并存储客户通过购物车输入的信息，然后再处理这些数据中的小问题。这与收入息息相关。你永远都不能告诉客户你的系统无法接收他们输入的信息。在这种情况下应选择可用性。
- **范例 2，提交订单：**当订单最终就位时，各种完成订单的服务会处理所有事务，你的系统不再直接与客户互动。在这种情况下应选择一致性。如果无法在某一项数据上取得一致性，可以先处理其他数据，将来再处理这项数据。因为该任务的现阶段处理工作仅是读取数据，所以也应该好好利用这一点。应在支持高性能读取操作的多个数据库中实现一致性。

因为需要根据服务类型决定 CAP 要素的取舍，所以你应该根据具体情况选择两个最适合实现的要素。而不应在创建系统时，以求全责备的心态处理 CAP 原则和可伸缩性。

Amazon.com 的 Actor 模型设计原则

在美国计算机协会的一次访谈中，Amazon.com 的首席技术官 Werner Vogels 对 Amazon.com 架构 [ACM-Amazon] 中的服务方向做出了下列评论：

在术语“面向服务”流行起来之前我们就通过这种方式处理服务。我们的服务方向处理方式是指将数据与处理数据的业务逻辑封装起来，只有通过公开的服务界面才能访问这些数据。不允许从服务的外部直接访问数据库，而且服务之间也不会共享数据。

前面介绍过 Actor 模型的特点，将数据与业务逻辑封装到一起并且不在服务之间共享数据是 Actor 模型的核心原则。

因为必须实现可伸缩性而且不能以亡羊补牢的方式实现，所以必须找到更好的方式处理网络对分布式系统产生的影响。

网络并非可伸缩系统面对的唯一挑战。

多线程技术的难点

通过 Intel Xeon Phi 服务器的研究报告，你肯定已经明白程序员编写多线程化

的软件是一种必然。然而，编写多线程化的软件并不是一件容易的事情，也不是一件许多程序员都擅长的事情。诚然，几乎每位程序员都学过使用线程的方式并尝试使用过它们（如在某些教授编程的网站上）。但是，又有几位程序员在正式的系统中真正使用过多线程技术呢？我从20世纪80年代中期就开始研究多线程技术，因此可以确切地告诉你，共享可变状态和互斥锁这些多线程问题非常难。

设计多线程应用程序时最困难的事情之一，可能是不必进行初始设计。设计方案就像随波逐流，我亲身经历过这种情况。我们使程序保持简单并易于理解，然后测试程序，程序运行正常，之后进行更多的测试，程序继续运行正常。万事大吉！

然后进行用户验收测试并听取用户的更改要求。获得意料之外的更改要求后，你曾经了解的多线程代码开始进行无法预料的调整和转变。这永远都不会是令人愉快的结果。在某个异常情况中，用户要求在某个异步操作的正中间输入精确的信息。你会发出“为什么你不一开始就告诉我这个要求呢”之类的感叹。这个新要求会从开头影响整个设计。交付日期飞速逼近，已经没有时间重新设计脆弱的多线程代码了。

多线程软件的另一个难点是实际运行环境。尽管你可以进行反反复复的测试，但也无法预测出软件在实际运行环境中遇到的所有情况。这是因为你可能不是使用实际的运行环境测试的程序。即使最优秀的测试实验室也无法与实际运行环境相比。用户可能会做这样的事情也可能会做那样的事情，网络可能出这样的情况也可能会出那样的情况，其他软件可能会出这样的影响也可能会出那样的影响，而且计算机也会在无法预料的情况下出故障。防患于未然仍旧很难做到。

说了这么多困难，我还没有提死锁、活锁等真正的多线程开发难题。请对比这些多线程开发中可能出现的问题。

- **死锁**：两个或多个线程相互等待对方释放指定的资源，因为无法获得必需的资源而都无法继续执行。这就形成了一种僵局。
- **活锁**：该情况与死锁类似，但活锁线程的状态会一直相互改变。这就像一条独木桥上来了两个彼此谦让的人，结果两人都无法过桥。
- **饥饿**：一些线程以正常方式执行任务，但是另一些线程一直无法获得CPU周期，因而处于饥饿状态。当将任务分配给一部分线程，而没有分配给另一部分线程时，可能会出现这种情况，这通常会导致出现低效代码（下一个问题）。
- **低效代码**：尽管锁起了作用，但代码设计不允许线程顺畅地运行。频繁地获取和释放锁会导致空耗CPU核心的周期。在大多数情况中，多线程代码以这样高的同步程度执行，会使它的效率与单线程程序一样。

- **伪共享**：这种情况与低效代码类似，但许多程序员都没有认识到和准备好解决这个问题。如前所述，现代 CPU 设计通过高速缓存，使应用程序的数据与 CPU 核心更加贴近。这就引入了高速缓存的分级模式（四核处理器通常拥有 L1、L2 和 L3 高速缓存），一个 CPU 核心会拥有 L1 和 L2 高速缓存。高速缓存由高速缓存行³构成，其中的数据都来自内存。在执行线程时，线程可以非常轻松地从一个高速缓存行读取数据。当线程 X 需要对高速缓存行中的某个变量执行操作时，线程 Y 可能需要对同一高速缓存行中的另一个变量执行操作。⁴如果线程 X 通过某种方式使这行高速缓存中的数据作废，那么使用同一高速缓存行的线程 Y 就会拥有非法的高速缓存数据。线程 Y 使用的高速缓存行就必须根据线程 X 执行的更改操作，通过内存刷新本身的数据，需要通过内存刷新数据的高速缓存行越多，多线程软件的运行速度就会越慢。尽管你可以将多线程代码设计得很好，但是由于伪共享问题 [False-Sharing]，CPU 核心通常还是会被空耗。

导致这些问题的原因是多线程软件本身难以推导，以及现代处理器可能与最理想的工作方式背道而驰。尽管我们理解当每个任务都是独立的操作并与指定系统状态对应时，我们会面对怎样的设计挑战，但还是难以预测出当这些任务同时运行时，它们彼此之间会以怎样的方式相互影响。这是因为我们预测不出处理器调度任务的时间。这也是多线程代码具有不确定性的原因。这还是多线程软件天生会导致线程之间存在临时耦合性和依赖性的原因。一直以来我们就被教导，应通过努力工作降低软件中各个对象之间的耦合性，但多线程编程模型和惯例的基础特性，却逼迫我们通过普通情况下无法理解的方式增加耦合性。

我们想要的是更简单的编写多线程软件的方式。如果临时耦合性是一个问题，那么我们就应该通过临时方式降低耦合性。使用队列管理由线程执行的各项工作，通常是推荐的处理方式。一个线程向队列中添加新的任务时，处理工作的线程（工人线程）会从队列中取出任务，如图 3.4 所示。因为所有工人线程都会从工作队列取出它们的下一个任务，所以这能够减少线程之间的联系。这样线程之间的耦合性就会更低了。

³ 通常一个高速缓存行的容量为64字节。有些硬件可能支持32字节或128字节的高速缓存行。

⁴ CircularQueue对象就是一个很好的例子，其中包含一个线程向高速缓存执行写入操作，而另一个线程从相同的高速缓存位置执行读取操作的情况。可以通过3种方式解决两个线程争用一条或多条高速缓存行的情况：（1）刷新高速缓存中的数据。（2）修改写入操作索引。（3）修改读取操作索引。

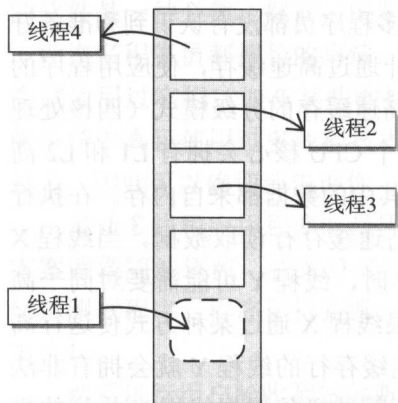


图 3.4 一个线程向队列中添加新的任务时，处理工作的线程会从队列中取出任务。

该处理方式中还有一个潜在的问题：这在需要使用资源的线程之间，生成了一种固有的竞争关系。如果某个线程能够更快速地获得任务，它就会总是先于其他工人线程锁住共享的队列，从而导致其他工人线程处于饥饿状态。甚至为工作队列添加工作的线程也会受到该线程的影响，而经常无法获得为该队列添加工作的机会。而且，也会出现多个任务使用同一个系统实体的情况，从而导致多个线程之间出现重叠的资源依赖性，进而导致出现死锁和活锁情况。即使能够避免死锁和活锁，被共享的资源本身也会导致争用资源的线程被阻塞，进而导致访问操作无法达到最佳标准，如图 3.5 所示。

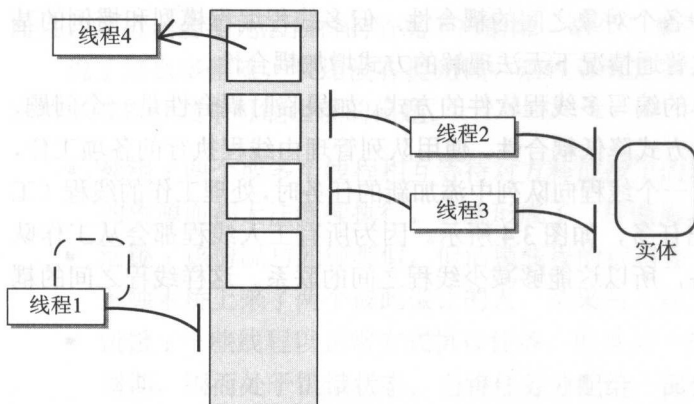


图 3.5 即使使用了工作队列处理方式，也可能在添加工作的线程和工人线程之间导致争用队列的情况。工人线程还有可能获得共享资源（如模型实体）重合的任务，从而导致访问资源的操作被阻塞。

可以使用 Java 的 `ConcurrentLinkedQueue` 等无锁队列。然而，即使使用无锁队列，如果该队列没有专门的防伪共享设计，也无法取得最理想的并发效果。

显然,即使使用 Java 和 C# 的标准线程化机制,需要解决的问题也不仅仅是两三个。令人遗憾的是,编译器和多线程工具都不能帮助我们消除伪共享导致的隐形系统开销。即使通过分析单个对象(如队列)可以防止在一组连续的存储空间中出现伪共享情况,但通常难以查明一个高速缓存行中可能含有哪些独立对象。

怎样才能将每个线程都设计得适合处理系统中任何类型的任务呢?当然,这是有可能做到的,但做到这一点是否很简单呢?此外,当前所有工人线程都使用基于轮询的队列访问操作,检索它们的下一项工作。我们是否可以改进这种模式,在了解到工人线程能够执行下一个任务时,将工作分配给它们?

尽管工作队列通常更适用于多线程设计,而且取得成功的可能性也高得多,但它仍旧不是理想的模式。研究 Intel Xeon Phi 服务器一段时间,你是否能够设计出一个基于工作队列的应用程序,使 288 个(或 277 个,为操作系统保留一个空闲 CPU 核心)CPU 核心一直满负荷工作?

摆在我们面前的问题是,怎样能够强制多个线程共享系统资源。单个的工作队列是一个较大的共享瓶颈。另一个潜在的共享瓶颈(死锁/活锁灾区)是通过共享方式使用模型实体。

通常,真正需要我们关心的,是当出现问题时怎样处理问题。当系统中正在运行的多个并发进程收到它们无法处理的数据、以不正确的方式执行操作或彻底崩溃时,你应该怎样处理呢?

正是由于并发和并行处理方式有这些潜在的副作用,所以主流的商务企业级解决方案大多会避免使用这类设计。这也是在大多数情况中多线程设计仅被前沿项目使用的原因,前沿项目必须压榨出每一分计算潜力并通过 CPU 核心和网络节点扩展规模。

我们应该能够更轻松地设计出更好的多线程应用程序。下面列举了一些与传统多线程设计模式相悖的断言,通过遵守这些断言可以使主流的企业级软件通过并发和并行设计获益。

- 如果因共享资源导致了问题(甚至导致出现网络分区),那么就应该停止共享资源、停止共享队列、停止共享领域模型实体并停止共享对它们执行操作的进程。
- 如果正式的运行环境会出现难以处理的突发情况,那么应尽可能地使我们的开发环境向其靠拢。
- 如果难以将线程设计得适合处理系统中所有需要处理的事项,应仅在有需要处理的任务时,为特定类型的任务分配专门处理这类任务的线程。
- 如果通过轮询方式分配工作的效率较低,就不要使用轮询方式分配工作。

将系统中的线程设计为在能够处理下一项工作时，才接收工作。只有当系统中的组件做好处理工作的准备并且有能够处理这项工作的可用线程时，组件才应对分配给它的工作做出回应，而不应通过主动轮询的方式获取要做的工作。

- 如果难以调度任务或者调度任务容易引发错误，可将这项工作交给擅长这方面的软件处理，你应将注意力集中在自己应用程序的开发工作上。
- 软件出故障是不可避免的。应通过预先制定的应对方案设计系统，使系统拥有容错性。

这些断言很有用处。然而，这些可靠的并发设计原则是否能够通过某种方式都实现呢？或者这仅是一种虚幻的万能灵药？是否存在真正实用的多线程软件开发技术，使我们能够推导系统、应对不断改变的情况并使系统拥有可伸缩性和容错性呢？

Actor模型的作用

Actor 系统可以帮助你同时利用多个处理器核心。为了使指定的 Actor 对象能够对它通过并发方式收到的消息做出回应，每个处理器核心都会被充分利用。调度器通常被用于将线程池中的线程分配给缓存了消息的 Actor 对象。在繁忙的系统中消息会不断进入许多 Actor 对象的消息缓存中，调度器会不停地为线程分配工作。一直以无阻塞方式使用线程，可以获得理想的、高效的并发处理系统。

这并不是说从一个 Actor 对象向另一个 Actor 对象发送消息，就像一个对象调用在同一个进程中的另一个对象中的方法那样迅速。这是两种完全不同的使用处理器核心的方式，后者代表几条极度优化的处理器指令，而前者需要用到大量的处理器指令。因此，Actor 模型并不是要在单个调用操作层级上取得突破。正如 Gul Agha 所提出的那样，顺序程序不擅长支持并发和并行处理方式的原因是，它们在处理并发情况时容易出现死锁 [Agha, Gul]。这也是我们使用 Actor 模型的最大原因。

然而，要高效地使用处理器和处理器的高速缓存并处理非常高的吞吐量，是否意味着我们必须无限地扩展服务器集群的规模呢？你可以像因特通公司、Google 和 Amazon.com 的分布式模型那样，在不使用成千上万台服务器的情况下设计应用程序。与传统应用程序服务器和第三方业务处理解决方案相比，Akka 集群会切实减少你的解决方案所需的服务器数量，这一定会使你感到惊讶。

下面介绍一个 WhitePages 公司的案例 [WhitePages]，该公司在美国是提供个人和企业联系信息的主要提供商。它每个月要为 5000 万个用户提供服务并处理

2 亿个搜索操作。该公司需要更换基于 Perl 和 Ruby 解决方案实现的系统。根据 WhitePages 公司提供的数据，它的一项服务需要使用 80 多台服务器每秒钟处理 400 个查询操作 [WhitePages]，而另一项服务需要使用 60 台服务器，这使纯硬件成本已经增长到一种夸张到荒谬的程度。这就像这些 Perl 和 Ruby 程序将 70% 的处理器资源用在了序列化数据和将数据解除序列化上。

通过使用 Scala 语言和 Akka 框架重新设计和实现该系统获得的结果令人惊讶。通过转换为响应处理方式，原来需要使用 60 台服务器的那项服务，迅速将所需服务器的数量降至 5 台。实际上，WhitePages 公司已经将他们所用的服务器数量降低了 15 倍 [WhitePages]。

如果你的应用程序需要在一个集群中扩展到 1,000 台或更多的服务器中，也不必担心，Akka 框架都可以满足你的需求。另一份体验报告展示了 Akka 框架，是怎样将程序扩展到 Google Compute Engine 云计算平台中的含有 2,400 个节点的集群中的 [Akka-Google]。将用 1,000 台服务器获得的效果与 WhitePages 公司使用 5 台服务器获得的效果进行比较。此处的要点是，不论你拥有多少台服务器，都应将它们的所有 CPU 核心和高速缓存充分利用起来，而不应通过效率较低的编程语言和技术使用它们。

下面让我们处理前面介绍过的 6 个断言，看看 Actor 模型是怎样回应它们的。这些解决方案中有些部分是融合到一起的。

1. **不共享**：Actor 模型专门规定 Actor 对象之间不共享任何可变状态。Actor 对象可以为其他 Actor 对象提供数据快照，但该数据对于接收者 Actor 对象来说必须是不可变的。此外，Actor 对象之间不共享队列。每个 Actor 对象都拥有本身独立的消息缓存，请参阅下面的第 3 点解决方案。基本上，按照预定设计运行的 Actor 对象不可能出现死锁和活锁情况。编写出低效 Actor 代码的少数几种可能性之一是，在访问某些资源（如硬盘）时出现阻塞情况。因此，除了少数几种 Actor 对象外，其他 Actor 对象永远都会避免出现阻塞情况。
2. **随机应变**：许多软件产品的故障是由它们的顺序实现方式无法适应变化和意料之外的环境问题导致的。因为在使用 Actor 模型的时候，你设计的是响应式系统，所以应尽量预测出所有意外情况。这不是要求你预测到所有可能出现的情况，因为任何人都无法做到这一点。然而，你可以通过监督机制为程序增加韧性，以便能够从容地处理意外情况，请参阅下面的第 6 点解决方案。
3. **Actor 模型**：使用不同类型的 Actor 对象代表应用程序中不同的主要概

念。将每个业务功能中特有的任务，分配给代表该业务领域的特定类型的 Actor 对象。应根据主要的业务功能精心划分 Actor 系统。请参阅《实现领域驱动设计》[IDDD] 中介绍集合和有界上下文的内容。

4. 通过消息指挥 Actor 对象：除了第3点解决方案外，Actor 对象还拥有用于暂时存储消息的消息缓存。当 Actor 对象能够处理下一条消息时，就会从消息缓存中取出下一条消息并处理它。只有当 Actor 对象能够理解消息、消息可以应用于 Actor 对象的当前状态并且调度器能够提供可用的线程时，Actor 对象才能对消息做出回应。

5. Actor 系统应提供调度功能：Akka 工具集自带了一系列调度器，通过配置这些调度器可以支持各种消息传递功能。如果你需要使用特殊的调度器，可以自己编写它们。然而，如果你想要优化某个预制的调度器，最终还需依靠 Akka 开发团队做到这一点。你只需将精力集中在自己的应用程序上，允许 Akka 框架发挥出它的最佳效果。

6. 韧性设计：不仅应尽可能根据软件产品的真正运行环境设计 Actor 对象模型，还应使 Actor 对象具有较高的适应性。如果你的系统遇到了导致软件产品出现故障的预料外情况，应使 Actor 对象能够适应新情况。如果在不进行调整和重新设计的情况下，无法处理预料外的情况，可通过 Actor 对象的监督机制将不兼容的消息类型稳妥地记录下来，并在将来避免尝试处理这类消息，恢复到之前能够正常运行的时间点。

要彻底坚持性能信念，请重新阅读第1章，透彻地了解 Actor 对象和 Actor 模型的最基础的特点。这些易于使用的工具提供了非常强大的功能。在使用 Actor 模型时，可以在设计中使用响应式的简单堆栈，并使得服务器集群中的每个处理器核心都充分发挥作用，这样你就能够获得高效的设计和高效的并发处理模式。

处理伪共享

我可以直言不讳地说，使用 Actor 模型和 Akka 框架的程序员永远都不必担心伪共享问题。然而，对于确实想要设计高性能系统的程序员来说，伪共享确实是一大心病。

如前所述，伪共享会导致多线程代码出现严重的隐形的性能问题。Intel 提出，目前避免伪共享的最佳手段是代码检测 [Intel-FalseSharing]。一旦检测出伪共享问题，可以使用字节填充技术，在高速缓存行边界上放置热数据（如 Circular-Queue 对象用于执行读取和写入操作的索引）。尽管这样做能够在单个对象中防

止出现伪共享问题，但是难以区分位于同一高速缓存行中的不同独立对象。

好消息是 Actor 模型能够提供帮助。每个 Actor 对象同一时刻只能由一个线程访问，永远不会出现两个线程共用同一个 Actor 实例及其内部数据的情况。任何时候 Actor 对象的任何状态转换都是通过执行单个线程实现的。因此，你不必劳心劳力地使用字节填充技术在高速缓存行边界上放置专用的字段。这为消除伪共享提供了最大的潜在帮助之一。

但还是会出现多个 Actor 对象或一个 Actor 对象与其他类型的对象，处于同一个高速缓存行中的情况。遗憾的是，出现这种情况的概率还比较高，因为 Akka 框架的 Actor 特征仅拥有两个对象引用字段。在大多数情况中这是一个优点，因为在 64 位架构中，一个 Actor 对象的默认存储系统开销仅为 16 字节。然而，一个容量为 64 字节的高速缓存行，能够存储 4 个多这样的对象。当然，具体的 Actor 对象的实际大小，还会根据在类声明中设置的字段类型和数量而各不相同。如果你在某个 Actor 对象中添加了 4 个引用字段，这些字段仅占用 32 字节。加上 Actor 对象 16 字节的默认存储系统开销，在 64 字节的高速缓存行中该 Actor 对象仅会占用 48 字节。如这个小例子所示，一个高速缓存行中很容易含有两个 Actor 实例，或者一个 Actor 对象和另一个其他类型的对象。

那么应怎样处理这种情况呢？强制所有 Actor 对象使用符合 64 字节容量的尺寸好像没有意义。这样做会白白耗尽应用程序的堆内存，大幅度限制你在单个 JVM 中创建 Actor 对象的数量。

即使如此，在大多数情况下你也至少能够根据实际情况识别出应用程序中的最热数据并对其进行调整。为什么要这样做呢？如果你知道某些 Actor 对象或相关对象（如 CircularQueue）参与了性能要求非常高的操作，就应该确保使这些对象能够独占一个或多个高速缓存行。要做到这一点，可以使用字节填充技术 [Nitsan Wakart]。使用该技术可以获得很好的效果，但需要经过反复试验。然而，由于 Actor 对象同一时刻只能由一个线程访问，所以与使用传统的多线程设计模式相比，你面对的问题也会更少。在某些情况中，Actor 对象和其他数据被随机存储在堆内存中的模式，也有助于避免高速缓存行中出现伪共享问题。然而，不对这种意外得到的帮助抱太大的期望。

遗憾的是，至少在编译器通过内部函数提供帮助前，你不得不依靠现有的设备解决伪共享问题 [Mechanical-Sympathy]。

设计模式

后面的一系列章节会介绍设计模式。这些设计模式既可以用于应用程序设计，

也可以用于应用程序整合。

- 第4章：通过 Actor 对象传递消息
- 第5章：消息通道
- 第6章：消息结构
- 第7章：消息路由
- 第8章：消息转换
- 第9章：消息端点
- 第10章：系统管理和基础结构

附录 A 是本书的最后一部分内容，介绍了供 C# 开发者在 .NET 平台上使用的 Actor 模型工具集 Dotsero。这为 C# 开发者提供了体验 Actor 模型和设计模式的途径。

第 4 章

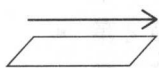
通过Actor对象传递消息

本章介绍 Actor 模型和 Akka 框架的基础设计模式。你不仅能够学到一些基本术语，还能学到设计单个 Actor 系统和将多个 Actor 系统整合到一起的方法。

- **消息通道**：是指消息生产者和消息消费者通信的方式，但这种通信方式不会使消息生产者和消息消费者有任何关联。在使用 Actor 模型时，从逻辑方面讲，消息通道就是 Actor 对象的消息缓存，是一种先入先出的队列。
- **消息**：是基于消息的系统的基础构成材料，Actor 模型中的消息与传统中间件系统中的消息是不同的。稍后会介绍设计消息、在 Actor 系统中封装消息和使用消息整合 Actor 系统的方式。
- **管道和过滤器**：在使用 Actor 模型时，使用管道和过滤器能够更从容、更轻松地解决问题。稍后会介绍使用消息执行加密、验证和删除操作的例子，展示通过消息将处理器链接到一起的方式。
- **消息路由器**：你在自己编写的 Actor 系统中会用到各种类型的消息路由器，其中包括有状态和无状态路由器、基于环境的和基于上下文的路由器、回环路由器和基于内容的路由器。稍后会介绍使用消息路由器的方式，和通过消息路由器查明消息接收者的方式。
- **消息译码器**：当消息进入和被送出系统时，会进行各种各样的转换。当与其他系统进行整合时，很可能需要使用消息译码器。只有收到与其兼容的数据，系统才能以适当的方式处理这些数据，因此很可能需要对数据格式进行转换。
- **消息端点**：在基于 Actor 的系统中，端点并不是特殊事物，它们仅是单个的 Actor 对象。在工作中你可以使用数百万数千万的消息端点。稍后会介绍在单个本地 Actor 系统中使用消息端点和使用消息端点整合其他系统的方式。

本章不仅将介绍基于 Actor 的消息传递功能的基础知识，还将介绍一些基本技巧，后面几章会详细介绍这些技巧。

消息通道



消息通道是消息生产者与消息消费者进行通信的方式，但这种通信方式不会使消息生产者和消息消费者有任何关联。在使用典型的消息传递机制时，消息生产者会使用消息消费者能够识别的标识符 / 名称创建一个通道。消息消费者打开该通道的接收端，做好处理消息的准备。一旦消息生产者发送了消息，消息消费者就能够接收到该消息。

在使用 Actor 模型时，这种工作模式稍有差异。尽管消息通道不是客观存在的事物，但你可以在逻辑上将它视为存在的事物。每个 Actor 对象都会通过某种方式提供本身的消息通道，通过这些通道接收和发送消息。尽管消息通道中还含有其他组件，但最能够代表该通道的就是 Actor 对象的消息缓存，如图 4.1 所示。尽管如此，对于收到的消息来说，Actor 对象的消息缓存基本上就是一个先进先出的队列。在使用 Actor 模型的情况下，消息通道不是由消息生产者创建的，而是由导致消息通道概念被生成的 Actor 对象创建的。

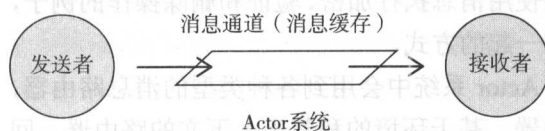


图 4.1 Actor 对象的消息缓存就是其逻辑的先进先出消息通道。

```
val processor = system.actorOf(Props[Processor], "processor")
// 消息 ProcessJob 被发送给名为 processor 的 Actor 对象
processor ! ProcessJob(1, 3, 5)
```

根据 Actor 模型的原则，消息生产者必须知道消息消费者的地址（实质上就是消息通道）。这意味着所有基于 Actor 的消息通道都是点对点型通道。即使在使用发布—订阅通道语义时，每个订阅者 Actor 对象仍旧会使用点对点通道与发布者进行通信。

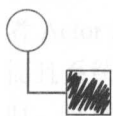
根据 Actor 模型的定义，消息至多只能向 Actor 对象发送一次。在许多情况中这种方式都能够取得不错的效果，尤其是在消息几乎总是能够顺利被送达的情况下。然而，如果出现某条消息没有成功送达目的地的情况，可以通过多种方式确保消息最终被送达目的地。例如，可以使发送者 Actor 对象监听对它发送消息

的回应，并设置回应时限。如果超出该时限后还没有收到回应，发送者 Actor 对象就应该重新发送这条消息。在出现崩溃故障的情况下，可能需要在重启发送者 Actor 对象后，才能重新发送这条消息。¹

在广泛使用事务的时代，这样做的风险好像有点高。毕竟，在企业商务中，完全接收不到某条消息（如金融交易消息）是无法接受的，对于发送完消息后就不再理会其效果的设计模式就更加不能接受了。在至多只能发送消息一次的环境中使用发送消息后不再理会其效果的设计模型时，谁能对产生的后果负责呢？在具有这些特点的情况下，Actor 模型适用于企业商务吗？

不必担心，可以通过多种方式确保指定的 Actor 对象最终能够接收到关乎业务成败的消息。在后面介绍确保送达、事务客户端 / Actor 对象和消息日志 / 存储等内容时，我会详细介绍 Akka 框架的坚持将消息送达的特性。

消息



消息是基于消息的系统的基础构建材料，基于消息的系统将操作和相关数据封装成消息实现通信。*Enterprise Integration Patterns* 一书 [EIP] 曾经介绍过消息传输系统通常会被标上中间件（被实现为消息服务器）的标签，而不是被视为 Actor 模型。消息中既含有系统级信息也含有应用程序级信息。消息由两个基本部分构成。

- 头部：供消息传输系统使用的信息。
- 主体：被传送的数据。

此外，使用中间件消息传输系统时，在进程之间用于装载消息的对象的类，通常是由消息传输系统提供的。例如，在 Java 和 .NET 系统中，Message 类型是由大多数中间件系统提供的，而且该类对象既拥有头部部分，也拥有主体部分（如图 4.2 所示）。

¹ 监督者可以监视崩溃故障并重启崩溃的 Actor 对象。还可以通过设置路由方案，使监督者先接收消息，在确保目标 Actor 对象正在运行的情况下，将消息转发给它。请参阅第 2 章中介绍监督机制的内容。

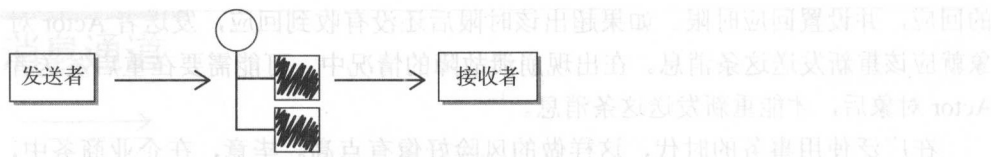


图 4.2 使用中间件工具的消息拥有头部和主体两个部分。

另一方面，在使用 Actor 模型时，消息仅含有第二个部分，即主体。换言之，消息仅是在 Actor 对象间传递应用程序级数据，如图 4.3 所示。如果存在供基础系统使用的任何头部信息，那么应用程序中的 Actor 对象不会看到这些信息。



图 4.3 Actor 模型中的消息仅含有应用程序数据。

因此，在 Actor 对象之间可以传递各种类型的消息，甚至包括标量值。因此，你可以将 Actor 对象设计为接收下列 Scala 类型的消息：

String

Int

Long

Float

Double

BigDecimal

Byte

Char

...

Any（代表由应用程序定义的特定的样本类类型）

因此，Akka 框架的 Actor 对象可以通过下列方式接收消息：

```
class ScalarValuePrinter extends Actor {
  def receive = {
    case value: String =>
      println(s"ScalarValuePrinter: received String $value")
    case value: Int =>
      println(s"ScalarValuePrinter: received Int $value")
    ...
  }
}
```

即使可以将标量值当作消息进行发送，也不意味着这是最佳选择。在某些情况中，这样做可能仅是在 Actor 对象之间传送简单数据。然而，当设计了完善的基于 Actor 的系统并将它们与原有系统整合到一起时，通常定义规范化消息模型（请参阅第 8 章）会更为有利。在使用 Scala 语言时，使用样本类可以高效地创建规范化消息模型中的消息类型。下面是一个命令消息的例子：

```
case class ExecuteBuyOrder(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
```

样本类有两大优点。首先，它们可专门用于支持 Scala 语言的模式匹配功能。其次，它们是创建不可变值类型的最简单方式之一。即使永远不应更改在 Actor 对象之间传递的消息是公认的原则，但将样本类用作适当的消息类型，能够确保每个消息实例保持不可变状态。

在 Scala 源代码模块中声明消息的类型时，应遵守几个常见的惯例。第一个惯例是，应将在源代码模块中由指定类型的 Actor 对象支持的所有消息，与接收者 Actor 对象的定义放在一起。当你编写的 Actor 系统主要使用本地 Actor 对象，而且系统中的所有 Actor 对象都能够通过软件包级的操作访问任何全局类型数据时，这样做能够取得最佳效果。这种源代码级的关联能够使为哪种 Actor 对象发送哪种消息的细节变得清晰，这样指定的接收者 Actor 对象就能够理解它收到的消息了。

第二个惯例是，将所有通用消息类型放置在一个 Scala 源文件中。当整合多个 Actor 系统中指定集合中的 Actor 对象，并统一定义规范化消息模型（一种在消息总线中推荐使用的处理方式）时，这样做可以取得最佳效果。下面是在交易消息总线示例中使用的一组命令消息和事件消息：

```
case class ExecuteBuyOrder(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
case class ExecuteSellOrder(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
case class BuyOrderExecuted(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
case class SellOrderExecuted(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
...
```

当需要跨多个远程 Actor 系统使用这些类型的消息，并且必须将这些消息类型部署在多个系统中，由这些系统中的接收者或发送者使用时，这样做尤为有用。

现在只需使用一个或少数几个源代码模块，就能够找到能够跨多个 Actor 系统的消息。这样做的缺点是代码可读性较差，即难以分辨出哪些 Actor 支持哪些消息类型。然而，可以使用一些文档和编程技巧，帮助你区分它们。

```
// 次序处理器消息
object OrderProcessorMessages {
  case class ExecuteBuyOrder(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
  case class ExecuteSellOrder(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
}
```

这段代码将消息样本类放在了 OrderProcessorMessages 对象中。这不仅可以起到文档的作用，而且还能用作命名空间技巧。要使用这些与处理次序关联起来的消息，就必须导入指定对象和该对象中包含的所有样本类。

```
import orders.processor.OrderProcessorMessages._
```

此外，还可以通过封装的特征（用于在进行匹配时支持特殊的防护功能）扩展所有消息。

```
// 次序处理器消息
object OrderProcessorMessages {
  sealed trait OrderProcessorMessage
  case class ExecuteBuyOrder(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
    extends OrderProcessorMessage
  case class ExecuteSellOrder(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
    extends OrderProcessorMessage
}
```

现在你可以强制规定对某一个样本类执行的匹配操作，必须能够应用于所有样本类。要做到这一点，必须使用嵌套的匹配结构。

```
class OrderProcessor extends Actor {
  import OrderProcessorMessages._
  def receive = {
    case message: OrderProcessorMessage => message match {
      case buy: ExecuteBuyOrder =>
        ...
      sender ! BuyOrderExecuted(...)
    }
  }
}
```

```
// 将会显示警告：
// "match may not be exhaustive... ExecuteSellOrder(_)"
}
}
}
```

这是一种防止开发者忘记考虑某些情况（如在更改设计时，将新的消息添加到消息传输协定中）的方式。Scala 编译器会对任何这类遗漏情况显示特定的匹配警告。

使用这种处理方式必定意味着需要在消息类型发生改变时，在每个系统中重新部署以样本类为基础的规范化消息模型。这将会背离当前企业级软件的整合原则（应根据架构整合系统，而不应根据类整合系统。应根据弱架构整合系统，而不应根据强架构整合系统²）。然而，在使用 Scala 语言和 Akka 框架开发软件时，这仍旧是一种常用的模式，而且在使用典型的消息传输中间件的情况下，你还需要面对因消息的类型发生改变所带来的挑战。

前面介绍过另一种处理方式——实现领域驱动设计 [IDDD]。在某些整合情况中，这种处理方式能够取得最佳效果。在领域驱动设计模式中，消息被称为通知。这种模式突出了基于文本的消息的作用，这类消息使用弱架构的 JSON 格式（JavaScript Object Notation），并且在收到并被解析时具有类型安全性。消息协定由公共语言传递 [IDDD]，应专门为 REST 自定义媒体类型设计公共语言。通过公共语言在整合传递消息的双方时起的作用，可以在不分发消息或不在接收端生成消息类型类的情况下交换消息。因此，你可能会发现使用格式指示器标识消息类型很有用，这样就能够同时支持多个不同的 Actor 系统。当使用消息桥、消息译码器、规整器等工具，在 Actor 系统和非 Actor 系统之间交换消息时，这样做尤为有用。

在使用消息总线时，另一种设计思路是在总线周围使用承载应用程序级消息的一系列总线级消息。

```
case class TradingCommand(
  commandId: String, command: Any)
case class TradingNotification(
  notificationId: String, notification: Any)
```

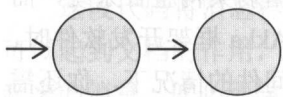
因此，特定的交易命令消息（如 ExecuteBuyOrder）和事件消息（如 BuyOrderExecuted），会分别通过 command 字段中的 TradingCommand 对象和

² 一个强架构的例子是使用严格的 XML 模式。一个弱架构的例子是使用 JSON 数据交换格式。

notification 字段中的 TradingNotification 对象，被放置在交易消息总线附近。通道适配器负责打包单个的 TradingCommand 和 TradingNotification 对象，还负责对这些对象进行拆包。总线级消息可能与中间件消息系统中的头部有点相似。它们之间既有相似的地方，也有不同的地方。毕竟，这是消息总线而不是 Akka 框架，而消息总线需要封装消息，请参阅后面对消息总线的介绍。

如 *Enterprise Integration Patterns*[EIP] 一书所述，除了命令消息和事件消息外，文档消息、请求—回应消息、消息序列和消息有效期都能被用于扩展基础消息。

管道和过滤器



在使用管道和过滤器架构时，你会通过将许多处理步骤链接到一起的方式构成处理过程。每个处理步骤都不会与其他处理步骤耦合。因此，当出现新需求时，所有处理步骤都会被重新安排或替换。

使用 Actor 模型构建管道和过滤器处理过程非常简单和轻松。图 4.4 展示了一个例子。一条订单消息进入系统，在为了完成购物操作处理完该条消息前，必须做一些预备工作。首先必须对这条订单消息进行解密，然后需要验证发送这条消息外部实体的资格，最后应确保这条订单消息不是之前收到消息的复制品。所有这些处理步骤都是由基于 Actor 的过滤器管理的，这些过滤器包括：Decrypter、Authenticator 和 Deduplicator。

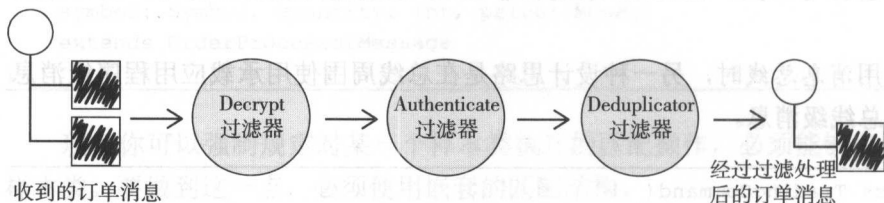


图 4.4 在管道和过滤器处理过程中 Actor 对象起过滤器的作用。

还有几个 Actor 对象也可以用作过滤器，其中包括一种消息端点 Actor 对象，这种对象会接收来自外部系统的订单消息，和一个用于在准备工作完成后处理该订单消息的 Actor 对象。下面列出了全部过滤器：

- OrderAcceptanceEndpoint
- Decrypter

- Authenticator
- Deduplicator
- OrderManagementSystem

在本例中，管道中的所有过滤器 Actor 对象（第一个除外），都收到了相同的标准的消息 `ProcessIncomingOrder`。此外，所有过滤器 Actor 对象（最后一个除外），都被作为构造器参数，传送给管道中的下一个过滤器。这两个设计方案都允许在管道中重新安排或彻底替换（根据需求添加新过滤器），除最后一个过滤器外的任意一个过滤器。

让我们先看看含有管道的应用程序：

```
object PipesAndFiltersDriver extends CompletableApp(9) {  
  val orderText = "(encryption)(certificate)<order id='123'>...</>  
  order>"  
  val rawOrderBytes = orderText.toCharArray.map(_.toByte)  
  val filter5 = system.actorOf(  
    Props[OrderManagementSystem],  
    "orderManagementSystem")  
  val filter4 = system.actorOf(  
    Props(classOf(Deduplicator), filter5),  
    "deduplicator")  
  val filter3 = system.actorOf(  
    Props(classOf(Authenticator), filter4),  
    "authenticator")  
  val filter2 = system.actorOf(  
    Props(classOf(Decrypter), filter3),  
    "decrypter")  
  val filter1 = system.actorOf(  
    Props(classOf(OrderAcceptanceEndpoint), filter2),  
    "orderAcceptanceEndpoint")  
  
  filter1 ! rawOrderBytes  
  filter1 ! rawOrderBytes  
  
  awaitCompletion()  
  println("PipesAndFiltersDriver: is completed.")  
}
```

过滤器 Actor 对象是以与它们在管道中的位置相反的顺序创建的，因此前一个创建的 Actor 对象会变成后一个创建的 Actor 对象在管道中后面的过滤器 Actor 对象。例如，`OrderManagementSystem` 对象会变为 `Deduplicator` 对象后

面的过滤器，以此类推。在运行该程序时，外部环境向该管道发送两条相同的 rawOrderBytes 消息，强行使 Deduplicator 过滤器检测出重复的订单消息并将其抛弃。

下面的代码从 OrderAcceptanceEndpoint 对象开始，使用管道顺序创建过滤器 Actor 对象：

```
class OrderAcceptanceEndpoint(nextFilter: ActorRef) extends Actor {
  def receive = {
    case rawOrder: Array[Byte] =>
      val text = new String(rawOrder)
      println(s"OrderAcceptanceEndpoint: processing $text")
      nextFilter ! ProcessIncomingOrder(rawOrder)
      PipesAndFiltersDriver.completedStep()
  }
}
```

如前所述，OrderAcceptanceEndpoint 是一个边界 Actor 对象，并且不会收到 ProcessIncomingOrder 消息。更确切地说，该 Actor 对象为每一条新进入系统的订单消息，通过 rawOrderBytes 消息创建 ProcessIncomingOrder 消息。前面介绍过，订单消息是经过加密的并且含有二进制的安全证书，因此接收者最终会以接收批量字节的方式接收它们。收到订单消息后，OrderAcceptanceEndpoint 过滤器会将这些字节封装到 ProcessIncomingOrder 消息中，并将该消息发送给它后面的过滤器 Actor 对象。

在管道中它后面的过滤器是 Decrypter，Decrypter 对象会收到一条 ProcessIncomingOrder 消息。

```
class Decrypter(nextFilter: ActorRef) extends Actor {
  def receive = {
    case message: ProcessIncomingOrder =>
      val text = new String(message.orderInfo)
      println(s"Decrypter: processing $text")
      val orderText = text.replace("(encryption)", "")
      nextFilter ! ProcessIncomingOrder(
        orderText.toCharArray.map(_.toByte))
      PipesAndFiltersDriver.completedStep()
  }
}
```

在这段普通的实现代码中，仅通过从订单消息中删除文本 (encryption)，模拟解密订单消息的操作。之后，Decrypter 过滤器会向它后面的过滤器

Authenticator 发送一条新的 ProcessIncomingOrder 消息（其中已经不含
有文本 (encryption)）。

```
class Authenticator(nextFilter: ActorRef) extends Actor {  
  def receive = {  
    case message: ProcessIncomingOrder =>  
      val text = new String(message.orderInfo)  
      println(s"Authenticator: processing $text")  
      val orderText = text.replace("(certificate)", "")  
      nextFilter !  
        ProcessIncomingOrder(orderText.toCharArray.map(_.toByte))  
      PipesAndFiltersDriver.completedStep()  
  }  
}
```

像处理 Decrypter 过滤器中的解密操作一样，我们仅在 Authenticator 对象中简单模拟了验证操作，将证书从消息中移除。完成这个步骤后，Authenticator 对象会向它后面的过滤器 Deduplicator 发送这条订单消息。

```
class Deduplicator(nextFilter: ActorRef) extends Actor {  
  val processedOrderIds = scala.collection.mutable.Set[String]()  
  
  def orderIdFrom(orderText: String): String = {  
    val orderIdIndex = orderText.indexOf("id=") + 4  
    val orderIdLastIndex = orderText.indexOf("'", orderIdIndex)  
    orderText.substring(orderIdIndex, orderIdLastIndex)  
  }  
  
  def receive = {  
    case message: ProcessIncomingOrder =>  
      val text = new String(message.orderInfo)  
      println(s"Deduplicator: processing $text")  
      val orderId = orderIdFrom(text)  
      if (processedOrderIds.add(orderId)) {  
        nextFilter ! message  
      } else {  
        println(s"Deduplicator: found duplicate order $orderId")  
      }  
      PipesAndFiltersDriver.completedStep()  
  }  
}
```

这个 Deduplicator 过滤器实际上没有跟踪复制的订单，它仅是简单地从

XML 文本中提取了每条订单消息的 ID。然后 Deduplicator 过滤器会尝试将订单 ID 添加到一个具有唯一性的 ID 集合（当然通常应使用可靠的永久存储机制处理该集合）中。如果该订单具有唯一性，那么它就会通过管道被传输。否则（即唯一性 ID 集合没有接受该订单的 ID，该订单仅是个副本），该订单消息就会被忽略。

最后，OrderManagementSystem 成为该管道的最后一个过滤器。

```
class OrderManagementSystem extends Actor {
  def receive = {
    case message: ProcessIncomingOrder =>
      val text = new String(message.orderInfo)
      println(s"OrderManagementSystem: processing unique order: $text")
      PipesAndFiltersDriver.completedStep()
  }
}
```

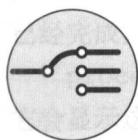
在本例中，OrderManagementSystem 过滤器表明只有当它收到具有唯一性的订单时，它才会处理订单。

下面是本例的输出结果：

```
OrderAcceptanceEndpoint: processing (encryption)(certificate)
<order id='123'>...</order>
OrderAcceptanceEndpoint: processing (encryption)(certificate)
<order id='123'>...</order>
Decrypter: processing (encryption)(certificate)
<order id='123'>...</order>
Decrypter: processing (encryption)(certificate)
<order id='123'>...</order>
Authenticator: processing (certificate)<order id='123'>...</order>
Authenticator: processing (certificate)<order id='123'>...</order>
Deduplicator: processing <order id='123'>...</order>
Deduplicator: processing <order id='123'>...</order>
Deduplicator: found duplicate order 123
OrderManagementSystem: processing unique order:
<order id='123'>...</order>
PipesAndFiltersDriver: is completed.
```

如 *Enterprise Integration Patterns*[EIP] 一书所述，虽然管道和过滤器框架中的所有执行处理操作的对象都被称为过滤器，但这些概念与内容过滤器和消息过滤器的概念不同。管道和过滤器架构中的过滤器不会过滤消息中的数据，而是用于在管道中执行处理操作。也就是说，Decrypter、Authenticator 和 Deduplicator 对象真正执行的是解密、验证和判断消息是否为复制品的操作。

消息路由器



管道和过滤器环境中通常会用到消息路由器。然而，消息路由器并不仅限于在这类环境中使用。各种消息路由器都有可能被用到，这些路由器包括：有状态和无状态路由器、基于环境的和基于上下文的路由器、回环路由器和基于内容的路由器。

路由器的常规功能是当收到消息时，路由器会检查消息的某个属性（或消息本身的某个状态）、环境的某个属性或上述所有这些元素，并通过符合技术或业务条件的消息通道分发当前的消息。换言之，消息路由器起的是分支机制的作用，与 if-else 代码块和分支语句非常相似，但使用独立的消息通道选择条件路径，如图 4.5 所示。

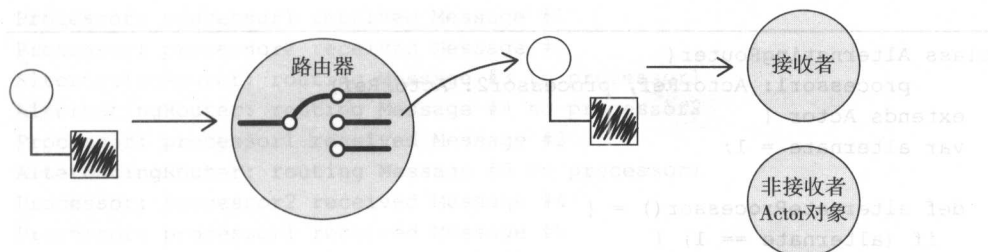


图 4.5 由消息路由器选择消息的最终接收者。

在使用 Actor 模型时，消息路由器本身就可以成为 Actor。路由器 Actor 对象会被放置在处理管道中适当的位置。当路由器收到消息后，会检查通过协定设置的条件并根据这些条件将消息分发给适当的 Actor 对象。这与有多少个 Actor 对象依靠该路由器转发消息无关，符合路由条件的 Actor 对象只会有一个。

下面让我们看一个路由器示例，该路由器会为两个处理器 Actor 对象分发消息。下面是含有该路由器的应用程序：

```
object MessageRouterDriver extends CompletableApp(20) {
  val processor1 = system.actorOf(Props[Processor], "processor1")
  val processor2 = system.actorOf(Props[Processor], "processor2")

  val alternatingRouter = system.actorOf(
    Props(classOf(AlternatingRouter), processor1, processor2),
```

```

"alternatingRouter")

for (count <- 1 to 10) {
  alternatingRouter ! "Message #" + count
}

awaitCompletion()

println("MessageRouter: is completed.")
}

```

Actor 类 Processor 的两个实例分别为 processor1 和 processor2，然后这段代码又创建了 AlternatingRouter 对象。之后该程序向 AlternatingRouter 对象发送了 10 条消息，每条消息都将当前消息计数用作本身的唯一标识。最后，该程序会在执行 20 个处理步骤后停止运行。每当一个处理步骤完成时，路由器（AlternatingRouter 对象）都会收到一条消息，而且两个处理器对象（processor1 和 processor2）其中之一也会收到一条消息。

AlternatingRouter 对象的代码非常简单。

```

class AlternatingRouter(
  processor1: ActorRef, processor2: ActorRef)
  extends Actor {
  var alternate = 1;

  def alternateProcessor() = {
    if (alternate == 1) {
      alternate = 2
      processor1
    } else {
      alternate = 1
      processor2
    }
  }

  def receive = {
    case message: Any =>
      val processor = alternateProcessor
      println(s" AlternatingRouter: routing $message to "
        ${processor.path.name}")
      processor ! message
      MessageRouter.completedStep()
  }
}

```

当路由器收到消息时，会调用它的 `alternateProcessor()` 方法。得到 `processor1` 或 `processor2` 对象其中之一的结果。路由器会通知程序一个步骤已经完成，然后将当前收到的消息转发给计算出的处理器 `Actor` 对象。

`processor Actor` 对象也非常简单。当 `processor Actor` 对象收到消息时，它会显示这条消息并通知程序另一个步骤已经完成。

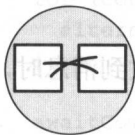
```
class Processor extends Actor {  
  def receive = {  
    case message: Any =>  
      println(s"Processor: ${self.path.name} received $message")  
      MessageRouter.completedStep()  
  }  
}
```

下面是路由器的输出结果和各种处理过程：

```
AlternatingRouter: routing Message #1 to processor1  
AlternatingRouter: routing Message #2 to processor2  
Processor: processor1 received Message #1  
Processor: processor2 received Message #2  
AlternatingRouter: routing Message #3 to processor1  
AlternatingRouter: routing Message #4 to processor2  
Processor: processor1 received Message #3  
AlternatingRouter: routing Message #5 to processor1  
Processor: processor2 received Message #4  
Processor: processor1 received Message #5  
AlternatingRouter: routing Message #6 to processor2  
Processor: processor2 received Message #6  
AlternatingRouter: routing Message #7 to processor1  
AlternatingRouter: routing Message #8 to processor2  
Processor: processor1 received Message #7  
AlternatingRouter: routing Message #9 to processor1  
Processor: processor2 received Message #8  
AlternatingRouter: routing Message #10 to processor2  
Processor: processor1 received Message #9  
Processor: processor2 received Message #10  
MessageRouter: is completed.
```

`AlternatingRouter` 路由器很简单，管道和过滤器处理过程中的其他类型的消息路由器也很简单。本例中的 `AlternatingRouter` 对象仅展示了一种可用的路由方式。请参阅第 7 章，详细了解更加专门化的路由方式。

消息译码器



使用消息译码器可以将消息中包含的数据转换为与本地环境兼容的数据，如图 4.6 所示。如 *Implementing Domain-Driven Design* 一书 [IDDD] 所述，在使用领域驱动的设计方式时，可以通过两种方式处理这种转换。

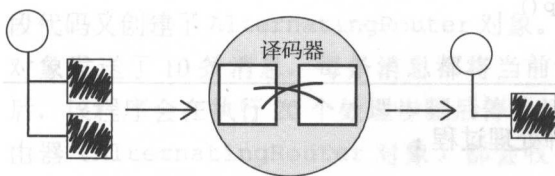


图 4.6 消息译码器将收到的消息转换为接收系统能够使用的数据。

- 在使用端口和适配器架构（也称为六角形架构）时 [IDDD]，可以使用应用程序边界的适配器 [GoF] 转换对应用程序的核心服务兼容参数影响较小的消息。也许该转换仅是将基于文本的消息转换为标量数据（API 能够接受的合法参数）。在这种情况下，端口的适配器是一种通道适配器，而且本质上是一种适宜的译码器。
- 在处理较复杂的转换情况时，通道适配器应该使用防腐化层 [IDDD]。某些数据可能比较复杂，需要使用更积极的译码方式。这可能需要使用多个适配器和特殊的译码器，将大量的原始标量数据转换为本地领域中的模型化概念。

你可以将第一种方式和第二种方式视为相同的转换方式，而两者的区别仅是它们含有的必要转换组件或多或少而已。可以将端口的适配器视为一种简单的防腐化层。然而，它们之间还有不少差异。在使用端口和适配器架构时，即使在转换过程较复杂的情况下，系统的边界上总是会存在端口的适配器。而且向较复杂的转换过程分配适用适配器的工作，也必然由系统边界上的端口适配器负责。

如 *Enterprise Integration Patterns* 一书 [EIP] 所述，可以使用各种工具完成转换过程。例如，使用可扩展样式表语言 (XSL) 可以高效地转换 XML 文本。还可以使用可视化转换工具，直观地以字段为单位将不同格式的数据对应起来。如果你使用弱架构（如 JSON），还可以使用基于分析程序的运行时转换方式，请参阅 *Implementing Domain-Driven Design* 一书 [IDDD]。有时还可以通过解除序列化

操作，将指定消息直接转换为本地对象或记录类型。³

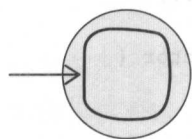
收到消息的类型（如命令消息、文档消息或事件消息），通常会决定必须执行的转换步骤的数量。例如，命令消息含有由接收系统指定的有效数据。收到的消息必须符合客户端对被请求的特定应用程序命令的约定。此外，命令消息的输出结果很可能是事件消息。

收到事件消息后需要执行的转换步骤比处理命令消息的转换步骤更多，但很可能仅会多一点而已。事件用于传达有限数量的信息，但足以清楚地告诉其他系统源系统之前出现的情况。消息的类型/名称本身就传达了许多事件的含义，其中通常会含有一个表明已发生情况的动词过去式。此外，在许多情况中，事件消息仅被用作标识符和标量数据。当接收系统转换事件消息时，事件消息会变成一条命令并被分派给本地应用程序服务。

另一方面，文档消息非常适合传输具有复杂和精细结构的大型有效数据负荷。因此，如果你知道需要使用消息封装比命令或事件更多的信息时，就应专门使用文档消息。

只要在 Actor 模型中使用消息译码器，整合的消息接收者就会成为消息端点 Actor 对象。在了解到转换过程会以异步方式完成的情况下，消息端点可以将另一个 Actor 对象用作主转换适配器。当数据格式较复杂、数据量较大并且需要使用用户可感知的时间范围执行转换过程时，这一点特别有用。

消息端点



在消息系统中，消息端点是指用于传输消息的消息通道两端的消息发送者和接收者。在使用 Actor 模型时，因为 Actor 对象既是消息的发送者也是消息的接收者，所以一般而言，Actor 对象就是消息端点，如图 4.7 所示。因此，基于 Actor 的系统中可能含有数千、数百万或数亿个消息端点。

³ 记录可以代表通用数据，如字典、映射或函数式语言中的特定记录类型。

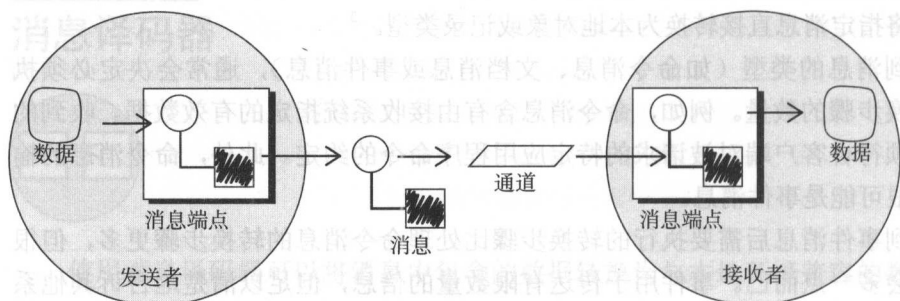


图 4.7 每个接收并处理消息的 Actor 对象都是消息端点。

然而，在进行整合时，你可能会使用通用化较低专门化较高的观点看待消息端点。在较特殊的情况中，可能需要在系统边界放置一个或多个消息端点 Actor 对象，为整合操作提供专用接口。也就是说，消息端点 Actor 对象允许在应用程序之间执行整合操作。不论系统中是否存在其他 Actor 对象扮演指定的应用程序角色，消息端点 Actor 对象都能够做到这一点。如果特定的应用程序 Actor 对象从专门的整合端点 Actor 对象那里收到消息，那么你就可以选择不将特定的应用程序 Actor 对象视为端点，而将它们视为特定的领域概念。

例如，如果有大量的网络零售商想要加入你的折扣系统，以便获取各种商品的报价。你可以创建一个报价曲线图类型的 Actor 对象(HighSierraPriceQuotes)，并将该 Actor 对象用作整合端点，部署到系统的边界。这个边界 Actor 对象应提供整合接口，为大量的零售商提供商品报价。

```
case class CalculateDiscountPriceFor(requester: ActorRef, ...)
...
class HighSierraPriceQuotes(discounter: ActorRef) extends Actor {
  val quoterId = self.path.name
  def receive = {
    case rpq: RequestPriceQuote =>
      discounter ! CalculateDiscountPriceFor(sender, rpq.retailerId,
                                             rpq.rfqId, rpq.itemId)
    case pricing: DiscountPriceCalculated =>
      pricing.requestedBy ! PriceQuote(quoterId, pricing.retailerId,
                                         pricing.rfqId, pricing.itemId,
                                         pricing.retailPrice,
                                         pricing.discountPrice)
    ...
  }
}
```

Actor 对象 HighSierraPriceQuotes 就是一个整合端点。它会从系统外部接

收 `RequestPriceQuote` 消息（由大量网络零售商发送的）。要满足这些请求，边界 `Actor` 对象 `HighSierraPriceQuotes` 应在 `Actor` 对象 `discounter` 被实例化时获得该对象的引用。`HighSierraPriceQuotes` 对象应该知道 `Actor` 对象 `discounter`（代表报价折扣计算器）能够处理 `CalculateDiscountPriceFor` 类型的消息。`discounter` 对象完成对指定零售商的报价折扣计算时，会对发送报价数据请求的 `HighSierraPriceQuotes` 对象回复 `DiscountPriceCalculated` 消息。

最后，当边界端点收到 `DiscountPriceCalculated` 消息时，`HighSierraPriceQuotes` 对象会向该消息的原始发送者（由 `pricing.requestedBy` 变量中的 `ActorRef` 引用指明）回复一条 `PriceQuote` 消息。因此，当 `HighSierraPriceQuotes` 对象成为边界端点时，`discounter` 对象就会成为指定领域中的 `Actor` 对象。即使如此，当通过普通方式推导时，还是可以将 `discounter` 对象视为一个端点。然而，当将该对象用作 `Actor` 对象时，`discounter` 对象起的作用远不仅仅是一个报价折扣计算器。

小结

本章介绍了使用 `Actor` 模型编写程序、应用程序设计模式和企业级软件整合的基础知识。还介绍了一些常用的术语。

本章介绍的编程模式还会在后面的章节中提到，后面各章会更详细地介绍它们。例如，后面会更详细地介绍各种消息路由器。

第 5 章

消息通道

上一章介绍了消息通道是一种基础通信机制，用于支持在 Actor 对象之间传递消息。本章会更加详细地介绍消息通道，其中包括基础的通道机制和几种通道类型。不同的通道类型适合应对不同的应用程序和整合挑战。

- **点对点通道**：这是 Actor 模型中必不可少的组成部分，因为它替代了面向对象系统中常规的方法调用模式。Actor 对象之间的通信自然而然地生成了一种点对点通道，因为发送消息的 Actor 对象必须知道接收消息的 Actor 对象的地址。这种模式会展示出消息次序，和 Actor 对象必须根据本身的当前状态做接收和拒绝消息的准备工作的原因。
- **发布—订阅通道**：Actor 对象通过发布—订阅通道可以将一条消息发送给多个 Actor 对象。实际上，Akka 框架既提供了本地发布—订阅通道功能，也提供了远程发布—订阅通道功能。
- **数据类型通道**：当接收消息的应用程序必须在不检查消息内容的情况下，接收指定数据类型的消息时，就应将独立的 Actor 对象创建为数据类型通道。
- **非法消息通道**：有时客户端会向 Actor 对象发送它无法识别或在当前状态下无法处理的消息。当出现这类情况时，收到消息的 Actor 对象可以将这些非法消息转发给非法消息通道，通过指定的应用程序方式处理它们。
- **死信通道**：当 Actor 系统无法将某条消息送达接收者，那么它就会将这条消息发送给死信通道。当将消息发送给已经停止并且不会再接收消息的 Actor 对象时，就会出现这种情况。
- **确保送达机制**：Actor 模型不会自带确保将消息送达的功能。它使用至多发送一次的消息传输机制。在大多数情况（如消息传输过程几乎不会受到其他因素影响、确保消息送达的系统开销过高）中，这种机制可以取得很好的效果。然而，有时需要确保 Actor 对象之间通信的可靠性。也就是说，不论遇到任何阻碍都必须将消息送达。这种模式展示了 Akka 框架中

的 Actor 对象能够将必须送达的消息可靠地传送给其他 Actor 对象。

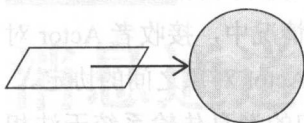
- **通道适配器**：当应用程序外围的 Actor 对象向应用程序内部的 Actor 对象发送消息时，需要使消息遵守内部协定。在这类情况中，接收者 Actor 对象能够在消息被通道适配器发送前，使消息遵守 Actor 对象之间的协定。
- **消息桥**：当需要整合两个应用程序时，如果它们的消息传输系统无法相互协作，那么就需要在这两个应用程序之间创建一座消息桥。支持 Java Message System (JMS) 的系统和基于 Akka 框架的 Actor 系统，与支持 RabbitMQ 的系统和基于 Akka 框架的 Actor 系统都是典型的例子。
- **消息总线**：这是一种较复杂的通道，但是在业务种类较多的企业中，这种通道会很有用。在为了提高竞争力，与商务伙伴整合的过程中，你可能会遇到各种各样的业务系统，从进货应用程序到销售应用程序等。尽管这些系统在不同的平台上运行并拥有各种各样的服务接口，但你仍然可以通过使用消息总线使这些系统在一起协同工作。

如 *Enterprise Integration Patterns* 一书 [EIP] 所述，可以在各种消息传输模式中使用这些通道，这些模式包括如下几项。

- **点对点**：点对点通道。
- **一对多**：发布—订阅通道。
- **数据类型识别**：数据类型通道。
- **非法和死信消息**：非法消息通道和死信通道。
- **故障耐受**：这是一种确保送达模式。也就是说，消息传输过程能够经得起任何系统故障的考验，这与 Akka 框架中 Actor 对象提供的常规韧性功能不同。
- **非消息传输客户端和自适应消息传输通道**：通道适配器和消息桥。
- **通信支柱**：消息总线。

这个消息传输模式列表表明，要在一个 Actor 系统中同时使用这些模式，需要投入相当多的精力进行思考和设计。尽管这些模式中有些概念极为简单，但是你必须理解所有这些模式的概念。在各种应用程序和整合情况中，以适当的方式使用这些通道才是关键所在。

点对点通道



在使用 Actor 模型时，所有 Actor 对象专用的消息通道都是点对点通道。这并不是说 Actor 模型不支持发布—订阅通道。订阅指定主题的消息 / 交换信息的 Actor 对象，能够通过点对点通道获得发布者发送的主题消息 / 交换信息。由此可见，使用点对点通道可以定义 Actor 模型的基础语义，如图 5.1 所示。与此类似，当 Actor 对象收到消息时，它还是一个由事件驱动的消费者。

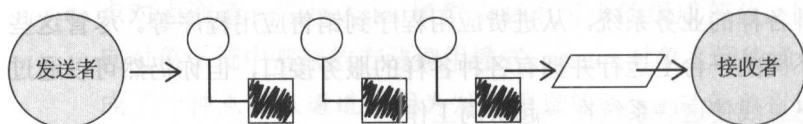


图 5.1 点对点通道中的 Actor 对象。

Actor 模型能够确保的功能之一，是以顺序方式传送消息。也就是说，默认情况下 Actor 对象的消息缓存是一种先入先出（FIFO）的通道。当消息沿着该通道到达时，就会以它们被发送的次序被接收。因此，如果 Actor 对象 A 向 Actor 对象 B 发送了一条消息，之后 Actor 对象 A 又向 Actor 对象 B 发送了一条消息，那么先发送的消息会先被 Actor 对象 B 收到。

```
// Actor 对象 A 内部的代码
actorB ! "Hello, from actor A!"
actorB ! "Hello again, from actor A!"
// Actor 对象 B 内部的代码
class ActorB extends Actor {
  var hello = 0
  var helloAgain = 0

  def receive = {
    case message: String =>
      hello = hello +
        (if (message.contains("Hello")) 1 else 0)
      helloAgain = helloAgain +
        (if (message.startsWith("Hello again")) 1 else 0)
      assert(hello > helloAgain)
  }
}
```

如果在该消息传输过程中引入第三个 Actor 对象 (C), 会出现怎样的情况呢? 现在 Actor 对象 A 和 C 都会向 Actor 对象 B 发送一条或多条消息。我们无法确保使 Actor 对象 B 先接收到哪一条消息, 因为 Actor 对象 A 可能先发送消息, 而 Actor 对象 C 也可能先发送消息。虽然如此, Actor 对象 A 发送的第一条消息永远都会先于 Actor 对象 A 发送的第二条消息被 Actor 对象 B 接收到, 而且 Actor 对象 C 发送的第一条消息永远都会先于 Actor 对象 C 发送的第二条消息被 Actor 对象 B 接收到。

```
// Actor 对象 A 内部的代码
actorB ! "Hello, from actor A!"
actorB ! "Hello again, from actor A!"

// Actor 对象 C 内部的代码
actorB ! "Goodbye, from actor C!"
actorB ! "Goodbye again, from actor C!"

// Actor 对象 B 内部的代码
class ActorB extends Actor {
  var goodbye = 0
  var goodbyeAgain = 0
  var hello = 0
  var helloAgain = 0

  def receive = {
    case message: String =>
      hello = hello +
        (if (message.contains("Hello")) 1 else 0)
      helloAgain = helloAgain +
        (if (message.startsWith("Hello again")) 1 else 0)
      assert(hello == 0 || hello > helloAgain)

      goodbye = goodbye +
        (if (message.contains("Goodbye")) 1 else 0)
      goodbyeAgain = goodbyeAgain +
        (if (message.startsWith("Goodbye again")) 1 else 0)
      assert(goodbye == 0 || goodbye > goodbyeAgain)
  }
}
```

注意, 在这段 Actor 对象 B (ActorB) 的实现代码中, 无法通过切实可行的

方式正确区分 hello 和 goodbye 消息的重要性等级。¹ 只能断言消息 hello 一定是 Actor 对象 A 发送的第一条消息, 而且其优先权高于消息 helloAgain, 以及消息 goodbye 一定是 Actor 对象 C 发送的第一条消息, 而且其优先权高于消息 goodbyeAgain。

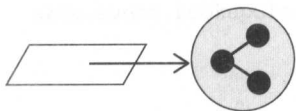
这意味着什么呢? Actor 对象必须做好准备工作, 以便根据它们的当前状态(由先前消息被接收的次序反映的)接收或拒绝消息。有时候延迟的消息也会被 Actor 对象接收, 但 Actor 对象必须谨慎地根据它过去的状态对延迟的消息做出回应。通过 Actor 对象的 become() 功能, 可以更从容地处理这类情况。

在使用 Akka 框架的远程处理工具 Akka Remoting 时, 还需要注意潜在的消息次序。当本地 Actor 对象通过 ActorSystem 对象使用 actorOf() 方法, 在远程节点中创建 Actor 对象时, 执行创建操作的请求是通过消息发送给远程节点的。本地 Actor 对象会收到从 RemoteActorRef 引用中提取出来的 ActorRef 引用。然而, 创建 Actor 对象的操作总是会以异步方式执行, 这意味着 RemoteActorRef 引用会先于它指向的远程 Actor 对象被创建。如果此时创建远程 Actor 对象的本地 Actor 对象向由 RemoteActorRef 引用指向的远程 Actor 对象发送消息, 那么当远程 Actor 对象被创建完成后, 该对象一定会收到这条消息。然而, 如果本地 Actor 系统中的其他 Actor 对象或第三方节点中的某个 Actor 对象, 想要通过 RemoteActorRef 引用向该远程 Actor 对象发送消息, 就无法确保在远程 Actor 对象被创建后向该对象发送消息。如果向 Actor 对象发送消息时该 Actor 对象还不存在, 那么任何消息都会丢失。

此处的要点是远程 Actor 对象的创建者, 只有在确定该远程 Actor 对象已经被创建完成后, 才能将 RemoteActorRef 引用赠予其他 Actor 对象。可以为本地 Actor 对象设置请求—回复协定, 强制被创建的远程 Actor 对象向它的父对象/创建者报告它本身的存在情况。也可以让父对象向远程 Actor 对象发送的第一条消息, 包含将会与该远程 Actor 对象协同工作的 Actor 对象的 ActorRef 引用。这会使新创建的远程 Actor 对象向它的协作 Actor 对象通报它本身的存在情况。这样做的缺点是必须了解它的协作 Actor 对象有哪些。在大多数设计情况中, 第一种方式更为实用, 有时也需要使用第二种方式。

¹ 有时被发送的消息不会仅有4条。即使可以预料到ActorB对象仅会收到4条消息, 但只有当收到全部这4条消息后, 才能断言消息hello和goodbye与消息helloAgain和goodbyeAgain具有相等的优先权, 在此之前ActorB对象只能处于等待状态。

发布—订阅通道



你可能是在接触观察者模式时 [GoF]，第一次了解发布—订阅模式的 [POSA1]。观察者模式和发布—订阅模式的基础设计原则，是将对事件消息感兴趣的对象与提供事件消息的对象隔离开。在观察者模式中，这两个角色被称为观察者和被观察者。在发布—订阅模式中，这两个角色被称为订阅者和发布者。

实际上，在评估典型的消息传输中间件系统时，可以将持久的和非持久的发布—订阅通道都视为自带功能。Actor 模型不是典型的消息传输系统，也不专门支持发布—订阅模式。Actor 模型着重支持的是点对点通道，但使用 Actor 模型提供的基础工具实现发布—订阅模式不会有任何问题。实际上，Akka 框架确实提供了几个内置的发布—订阅工具。图 5.2 展示了使用 Akka 框架中的 EventBus 类创建发布—订阅通道的方式。

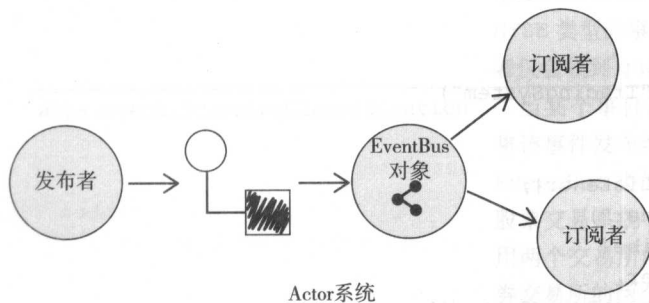


图 5.2 Actor 对象可以通过 EventBus 对象，向多个订阅者发布消息。

本地事件流

Akka 框架内置的几个发布—订阅通道，其中之一使用了 EventBus 特征。标准的 EventBus 实例称为事件流，通过本地 Actor 系统的 eventStream 属性可以获得该对象。这种标准的 EventBus 对象可以在本地 JVM 中使用，而且使你能够很轻松地注册订阅者和使发布者发布事件。

通过 EventBus 对象可以传送 Akka 框架的标准系统消息，如下所示创建订阅者非常简单。本例使用了死信通道例子中使用的代码：

```
val system = ActorSystem("TradingSystem")
val sysListener =
    system.actorOf(
        Props[SystemListener],
        "sysListener")
system.eventStream.subscribe(
    sysListener,
    classOf[akka.actor.DeadLetter])
```

使用这类事件流，你还可以订阅标准日志事件。即便如此，标准事件也不是应用程序事件，而且你需要创建能够满足特定应用程序需求的本地发布—订阅读道。一个解决方案是通过使用应用程序特有的分类器注册订阅者，在标准事件流中附加信息。

```
case class Money(amount: BigDecimal) { ... }
case class Market(name: String)
case class PriceQuoted(
    market: Market,
    ticker: Symbol,
    price: Money)
...
val system = ActorSystem("TradingSystem")
val quoteListener =
    system.actorOf(
        Props[QuoteListener],
        "quoteListener")
system.eventStream.subscribe(
    quoteListener,
    classOf[quotes.PriceQuoted])
```

该解决方案的后半部分，是将 PriceQuoted 事件发布给通道。

```
system.eventStream.publish(PriceQuoted(ticker, price))
```

在本例中，简单的样本类 PriceQuoted 就是一个分类器，它决定了哪一类订阅者可以接收哪一类事件。因此，看起来没有必要为 EventBus 类添加自定义代码。

然而，为什么还要创建自定义的 EventBus 类呢？一个原因可能是为了使注册的订阅者支持指定的事件类型，本质上就是实现指定的功能。自定义的 EventBus 类必须扩展 EventBus 特征（如 ActorEventBus），而且还要混入分类特征。表 5.1 展示了一些实用的分类。

表 5.1 分类特征

分类	描述
<code>akka.event.LookupClassification</code>	通过匹配指定的事件类型，支持简单的查询操作。例如，可以使用同一个 <code>EventBus</code> 对象支持 <code>quotes</code> 和 <code>priortityQuotes</code> 通道。如果 <code>Actor</code> 对象订阅了 <code>quotes</code> 通道，那么该 <code>Actor</code> 对象仅会收到发布给该通道的事件。如果 <code>Actor</code> 对象订阅了 <code>priortityQuotes</code> 通道，那么该 <code>Actor</code> 对象仅会收到发布给该通道的事件
<code>akka.event.SubchannelClassification</code>	通过匹配事件的类型和子类型，支持子通道层次结构。例如，可以通过同一个 <code>EventBus</code> 对象使 <code>quotes</code> 通道能够接收 <code>quotes/NYSE</code> 和 <code>quotes/NASDAQ</code> 子类型的事件。发布者可以发布 <code>quotes</code> 、 <code>quotes/NYSE</code> 和 <code>quotes/NASDAQ</code> 类型的事件，订阅者也可以订阅这些类型的事件。 <code>quotes</code> 通道的订阅者会收到 <code>quotes</code> 、 <code>quotes/NYSE</code> 和 <code>quotes/NASDAQ</code> 类型的事件。 <code>quotes/NYSE</code> 通道的订阅者仅会收到 <code>quotes/NYSE</code> 类型的事件。 <code>quotes/NASDAQ</code> 通道的订阅者仅会收到 <code>quotes/NASDAQ</code> 类型的事件
<code>akka.event.ScanningClassification</code>	当某个事件落入两个或多个通道，并且需要将事件发布给所有合法订阅者，因而需要扫描 <code>EventBus</code> 类的全部分类时，应混入该特征。在股票交易系统中，尽管一个股票报价不会同时使用两个交易所的报价，但有可能同时使用一个证券交易所的两个金融参数

下面让我们使用表 5.1 介绍的 `SubchannelClassification` 分类创建一个 `QuotesEventBus` 类。这需要使用一些样本类提供支持，并将自定义的 `EventBus` 特征混入 `SubchannelClassification` 分类。

```
case class Money(amount: BigDecimal) {
  def this(amount: String) =
    this(new java.math.BigDecimal(amount))

  amount.setScale(4, BigDecimal.RoundingMode.HALF_UP)
}

case class Market(name: String)

case class PriceQuoted(
```

```

    market: Market,
    ticker: Symbol,
    price: Money)

class QuotesEventBus
    extends EventBus
    with SubchannelClassification {
    type Classifier = Market
    type Event = PriceQuoted
    type Subscriber = ActorRef

    protected def classify(event: Event): Classifier = {
        event.market
    }

    protected def publish(
        event: Event,
        subscriber: Subscriber): Unit = {
        subscriber ! event
    }

    protected def subclassification =
        new Subclassification[Classifier] {
        def isEqual(
            subscribedToClassifier: Classifier,
            eventClassifier: Classifier): Boolean = {
            eventClassifier.equals(subscribedToClassifier)
        }

        def isSubclass(
            subscribedToClassifier: Classifier,
            eventClassifier: Classifier): Boolean = {
            subscribedToClassifier.name.startsWith(eventClassifier.name)
        }
    }
}

```

支持 QuotesEventBus 类的样本类有 3 个: Money、Market 和 PriceQuoted。创建自定义的 EventBus 类需要声明几种类型: 分类器、事件和订阅者, 在本例的 SubchannelClassification 分类中, 这些类型被设置为 Market、PriceQuoted 和 ActorRef。

```

class QuotesEventBus
    extends EventBus

```

```

with SubchannelClassification {
  type Classifier = Market
  type Event = PriceQuoted
  type Subscriber = ActorRef
  ...

```

classify() 函数会回应 / 返回事件的分类器, 在本例中由 PriceQuoted 事件的 market 属性代表。这就是将指定的 Market 分类器或分类与指定的订阅者对应起来的方式。

subclassification 对象 (Subclassification 实例) [Classifier] 提供了两种与指定 Market 分类器匹配的方式, 分别为 isEqual() 和 isSubclass() 函数。如果 isEqual() 函数确定了精确匹配的 Market 分类器, 那么订阅者就会收到 PriceQuoted 事件。

也可以使用 isSubclass() 函数匹配分类。该函数的第一个参数是 subscribedToClassifier, 它代表被指定订阅者订阅的 Market 分类器。该函数的第二个参数是 eventClassifier, 它代表与被发布的 PriceQuoted 事件关联的 Market 分类器。这个 isSubclass() 函数的设计非常简单, 为了确定正确的分类, 它允许 subscribedToClassifier 参数代表的分类的名称与 eventClassifier 参数代表的分类的名称部分或完全相同。因此, 使用两个完全相同的分类也是合法的。

```

...
def isSubclass(
  subscribedToClassifier: Classifier,
  eventClassifier: Classifier): Boolean = {
  subscribedToClassifier
    .name
    .startsWith(eventClassifier.name)
}
...

```

为了测试 QuotesEventBus 类, 我编写了一个简单的程序 (SubClassificationDriver)。

```

object SubClassificationDriver extends CompletableApp(6) {
  val allSubscriber =
    system.actorOf(
      Props[AllMarketsSubscriber],
      "AllMarketsSubscriber")

```



```

val nasdaqSubscriber =
  system.actorOf(
    Props[NASDAQSubscriber],
    "NASDAQSubscriber")
val nyseSubscriber =
  system.actorOf(
    Props[NYSESubscriber],
    "NYSESubscriber")

val quotesBus = new QuotesEventBus

quotesBus.subscribe(allSubscriber, Market("quotes"))
quotesBus.subscribe(nasdaqSubscriber, Market("quotes/NASDAQ"))
quotesBus.subscribe(nyseSubscriber, Market("quotes/NYSE"))

quotesBus.publish(PriceQuoted(Market("quotes/NYSE"),
  Symbol("ORCL"), new Money("37.84")))
quotesBus.publish(PriceQuoted(Market("quotes/NASDAQ"),
  Symbol("MSFT"), new Money("37.16")))
quotesBus.publish(PriceQuoted(Market("quotes/DAX"),
  Symbol("SAP:GR"), new Money("61.95")))
quotesBus.publish(PriceQuoted(Market("quotes/NKY"),
  Symbol("6701:JP"), new Money("237")))

awaitCompletion
}

```

第一步是创建 3 个 Actor 对象。一个 Actor 对象应通过指定的 Market("quotes") 分类器订阅所有股票报价。另一个 Actor 对象应通过指定的 Market("quotes/NASDAQ") 分类器，仅订阅纳斯达克股票报价。最后一个 Actor 对象应通过指定的 Market("quotes/NYSE") 分类器，仅订阅纽约证券交易所的股票报价。下面是这三个 Actor 类的实现代码：

```

class AllMarketsSubscriber extends Actor {
  def receive = {
    case quote: PriceQuoted =>
      println(s"AllMarketsSubscriber received: $quote")
      SubClassificationDriver.completedStep
  }
}

class NASDAQSubscriber extends Actor {
  def receive = {
    case quote: PriceQuoted =>
      println(s"NASDAQSubscriber received: $quote")
  }
}

```

```

        SubClassificationDriver.completedStep
    }
}

class NYSESubscriber extends Actor {
    def receive = {
        case quote: PriceQuoted =>
            println(s"NASDAQSubscriber received: $quote")
            SubClassificationDriver.completedStep
    }
}

```

创建好这 3 个 Actor 类后，应用程序 SubClassificationDriver 就能够发布事件了。该程序向 QuotesEventBus 对象发布了 4 个 PriceQuoted 事件，下面是 3 个 Actor 对象的输出结果：

```

AllMarketsSubscriber received:
PriceQuoted(Market(quotes/NYSE), 'ORCL, Money(37.84))
NYSESubscriber received:
PriceQuoted(Market(quotes/NYSE), 'ORCL, Money(37.84))
AllMarketsSubscriber received:
PriceQuoted(Market(quotes/NASDAQ), 'MSFT, Money(37.16))
NASDAQSubscriber received:
PriceQuoted(Market(quotes/NASDAQ), 'MSFT, Money(37.16))
AllMarketsSubscriber received:
PriceQuoted(Market(quotes/DAX), 'SAP:GR, Money(61.95))
AllMarketsSubscriber received:
PriceQuoted(Market(quotes/NKY), '6701:JP, Money(237))

```

Actor 对象 AllMarketsSubscriber 收到了全部 4 个事件，而 Actor 对象 NYSESubscriber 和 NASDAQSubscriber 都仅收到了一个事件。

分布式发布—订阅通道

Akka 框架中的第二种内置发布—订阅通道，专门用于 Akka 集群。这种发布—订阅通道提供了跨集群节点，根据主题向订阅者发布消息的功能。这种通道还支持在发送者不知道接收者在集群中的具体位置的情况下，向集群中的单个 Actor 对象发送消息，以及向分布在多个节点中的一个 Actor 对象发送消息。这是一种经过修改的发布—订阅通道，支持通过低耦合度方式直接发送消息。

Akka 框架提供了支持这种集群化发布—订阅功能的 Actor 类 DistributedPubSubMediator，如图 5.3 所示。这个起中间人作用的 Actor 对象，必须在所有参与指定发布—订阅主题或发送者—接收者协作的节点上启动。尽管这个中间人

Actor 对象能够通过多个独立的发布者和接收者逻辑分组，支持任意数量的主题，但最好为每个发布-订阅逻辑分组启动专用的 DistributedPubSubMediator 对象。例如，你可以为 bids 主题启动一个中间人 Actor 对象，为 sold 主题启动另一个中间人 Actor 对象，每个中间人 Actor 对象专门负责一组订阅者。

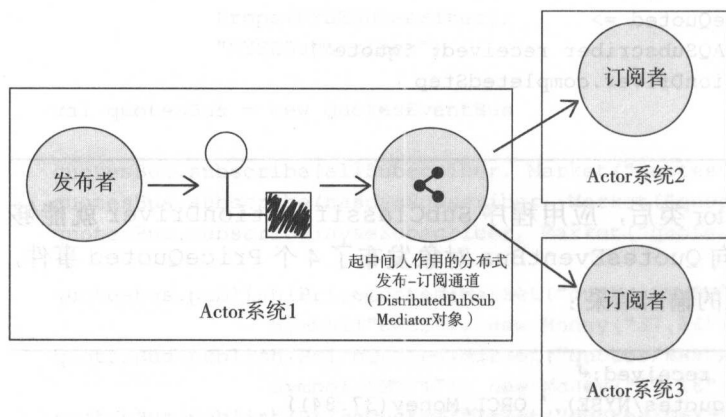


图 5.3 使用 DistributedPubSubMediator 对象可以向远程 Actor 系统中的订阅者发布消息。

可以像启动普通 Actor 对象一样启动 DistributedPubSubMediator 对象，也可以像管理普通 Actor 对象一样使用 DistributedPubSubExtension 扩展类管理 DistributedPubSubMediator 对象。当使用该扩展类时，在同一个集群中就只能使用一个 DistributedPubSubMediator 对象。如果你选择使用这个扩展类，就必须根据你自己的需要配置该扩展类。下面展示了一些外部配置属性，这些属性是标准的默认配置，还有一行用于使 Akka 框架自动运行 DistributedPubSubExtension 扩展类的代码：

```
akka.contrib.cluster.pub-sub {
  name = distributedPubSubMediator
  role = ""
  routing-logic = random
  gossip-interval = 1s
  removed-time-to-live = 120s
  max-delta-elements = 3000
}

akka.extensions = ["akka.contrib.pattern.DistributedPubSubExtension"]
```

最后一行代码会命令 Akka 框架运行 DistributedPubSubExtension 扩展类，这会创建一个名为 distributedPubSubMediator 的 Actor 实例。表 5.2 介

绍了 DistributedPubSubExtension 扩展类的所有配置属性。其中的描述部分详细介绍了中间人 Actor 对象的重要工作方式。

表 5.2 DistributedPubSubExtension 扩展类的属性

属性	描述
name	当通过 Actor 系统创建 Actor 对象时,使用该属性可以设置 Actor 对象的名称。如果将该属性设置为 distributedPubSubMediator,就会创建出 /user/distributedPubSubMediator 对象
role	该属性用于设置角色,设置了该属性后,就只能在拥有相同角色名称的集群节点中启动中间人 Actor 对象。如果没有设置该属性(即将该属性设置为 ""),就可以在集群中所有的节点上启动中间人 Actor 对象
routing-logic	当使用 Send 方法(而不是 Publish 方法)向一个集群节点中的多个 Actor 对象发送消息时,使用该属性可以设置路由策略。可设置的路由策略包括:random(随机)、roundrobin(轮转循环)、consistent-hashing(一致性散列)和 broadcast(广播)
gossip-interval	每个中间人 Actor 对象都会有一张已注册订阅者的名单。某个节点上的中间人 Actor 对象注册了订阅者后,必须向在其他节点上运行的中间人 Actor 对象通报这些订阅者的情况。中间人 Actor 对象会使用 gossip 协议与其他节点中的中间人 Actor 对象交流这些情况。gossip-interval 属性用于设置不同节点上中间人 Actor 对象相互通信的时间间隔
removed-time-to-live	实际上,中间人 Actor 对象的已注册订阅者名单中任何已经停止运行的订阅者,和通过其他方式被添加了删除标记的订阅者,需要经过指定的时间才会被真正从这个名单中移除。removed-time-to-live 属性用于设置该时间
max-delta-elements	在 gossip-interval 属性设置的时间间隔中,如果中间人 Actor 对象的已注册订阅者名单中增加了条目,那么这些增量情况就会被发送给其他节点中的中间人 Actor 对象。max-delta-elements 属性用于设置在 gossip-interval 时间间隔中,允许传输的增量情况的最多次数。余下的和新出现的增量情况,都会在下一个 gossip-interval 时间间隔中被传输

实际上,这些属性不是用于配置 DistributedPubSubExtension 对象的。更确切地说,DistributedPubSubExtension 对象只是加载这些属性并使用这些属性创建中间人 Actor 对象。

```
class DistributedPubSubExtension(system: ExtendedActorSystem)
  extends Extension {
```

```

private val config =
    system.settings.config.getConfig(
        "akka.contrib.cluster.pub-sub")
private val role: Option[String] =
    config.getString("role") match {
        case "" => None
        case r => Some(r)
    }

def isTerminated: Boolean =
    Cluster(system).isTerminated ||
    !role.forall(Cluster(system).selfRoles.contains)

val mediator: ActorRef = {
    if (isTerminated)
        system.deadLetters
    else {
        val routingLogic =
            config.getString("routing-logic") match {
                case "random" =>
                    RandomRoutingLogic()
                case "round-robin" =>
                    RoundRobinRoutingLogic()
                case "consistent-hashing" =>
                    ConsistentHashingRoutingLogic(system)
                case "broadcast" =>
                    BroadcastRoutingLogic()
                case other =>
                    throw new IllegalArgumentException(s"...")
            }
        val gossipInterval =
            Duration(config.getMilliseconds(
                "gossip-interval"),
                MILLISECONDS)
        val removedTimeToLive =
            Duration(config.getMilliseconds(
                "removed-time-to-live"),
                MILLISECONDS)
        val maxDeltaElements =
            config.getInt("max-delta-elements")
        val name = config.getString("name")
        system.actorOf(DistributedPubSubMediator.props(
            role, routingLogic,
            gossipInterval, removedTimeToLive,
            maxDeltaElements,
            name)
        )
    }
}

```

DistributedPubSubExtension 对象会使用 ActorSystem 对象中的 actorOf() 函数创建 DistributedPubSubMediator 对象。注意,即使你不使用扩展对象管理中间人 Actor 对象,也需要使用相同的方式创建中间人 Actor 实例。然而,在这种情况下,需要将合法的 Props 实例用作 ActorSystem 对象中的 actorOf() 函数的参数。DistributedPubSubMediator 伴生对象提供了一个便捷的辅助方法,该方法专门用于达到该目的。

```
object DistributedPubSubMediator {

  def props(
    role: Option[String],
    routingLogic: RoutingLogic = RandomRoutingLogic(),
    gossipInterval: FiniteDuration = 1.second,
    removedTimeToLive: FiniteDuration = 2.minutes,
    maxDeltaElements: Int = 3000): Props =
    Props(classOf[DistributedPubSubMediator],
          role, routingLogic, gossipInterval,
          removedTimeToLive, maxDeltaElements)

  ...
}
```

将这些默认的参数值与前面的配置属性做比较,我们可以看到伴生对象使用了相同的标准值。通过使用所有可用的默认参数,可以创建下面这样的中间人 Actor 对象:

```
val mediator = system.actorOf(
  DistributedPubSubMediator.props(None),
  "bidsPubSubMediator")
```

这段代码创建了一个新的中间人 Actor 对象,没有为该对象分配角色(设置 role 属性),将该对象的名称设置为 bidsPubSubMediator,同时会对 routingLogic、gossipInterval、removedTimeToLive 和 maxDeltaElements 属性使用标准的默认值。不为中间人 Actor 对象设置角色意味着,可以在集群中的任意一个节点上启动中间人 Actor 对象,并且使中间人 Actor 对象能够在其运行的节点上支持一条或多条发布—订阅通道。

下面介绍 DistributedPubSubMediator 对象能够执行的操作。DistributedPubSubMediator 对象主要能够执行 3 种操作,每种操作都是通过向该中间人 Actor 对象发送指定类型的消息执行的。表 5.3 介绍了这些操作。

表 5.3 DistributedPubSubMediator 对象的操作

消息类型	描述
DistributedPubSubMediator. Send	<p>如果你需要在集群中的多条路由线路中选择其中一条传输消息,可使用 Send 消息</p> <p>消息样本类 Send 接收 3 个初始化器: path、message 和 localAffinity。message 初始化器是最终会被发送的消息,而且会被发送给集群中唯一符合 path 初始化器的 Actor 对象。如果 message 初始化器的接收者 Actor 对象的名称为 bidProcessor,那么标识该 Actor 对象的 path 初始化器就一定是 /user/bidProcessor,而且必定不是通过 ActorSystem 对象的名称或地址预先配置的。因此,如果集群中有多个 Actor 对象(但至多仅有一个 Actor 对象正在活动的 Actor 系统中运行)的路径与 path 初始化器相符,那么 message 初始化器最终会根据初始化中间人 Actor 对象的 routingLogic 参数,仅被发送给集群中某个 Actor 系统中的一个 Actor 对象</p> <p>当 localAffinity 初始化器的值为 true 时,如果本地系统中的某个 Actor 对象的路径匹配指定的 path 初始化器,中间人 Actor 对象会尝试将 message 初始化器发送(确切地说是转发)给本地系统中的 Actor 对象。如果本地系统中没有哪个 Actor 对象的路径匹配指定的 path 初始化器,message 初始化器会根据初始化中间人 Actor 对象的 routingLogic 参数,仅被发送给集群中某个 Actor 系统中的一个 Actor 对象</p>
DistributedPubSubMediator. SendToAll	<p>如果你想向集群中所有路径与 path 初始化器匹配的 Actor 对象广播消息,可使用 SendToAll 消息</p> <p>消息样本类 SendToAll 接收 3 个初始化器: path、message 和 allButSelf。message 初始化器是最终会被发送的消息,而且会被发送给集群中所有符合 path 初始化器的 Actor 对象。因为在 Actor 对象的路径与 path 初始化器匹配的情况下,一个 Actor 系统中仅会存在一个符合条件的 Actor 对象,所以该消息至多仅会被发送给每个 Actor 系统中的一个 Actor 对象。如果 message 初始化器的接收者 Actor 对象的名称为 bidRelay,那么标识该 Actor 对象的 path 初始化器就一定是 /user/bidRelay,而且必定不是通过 ActorSystem 对象的名称或地址预先配置的</p>

续表

消息类型	描述
	<p>当 allButSelf 初始化器的值为 false（默认值）时，如果本地系统中的某个 Actor 对象的路径匹配指定的 path 初始化器，中间人 Actor 对象会尝试将 message 初始化器发送（确切地说是转发）给本地系统中的 Actor 对象。如果 allButSelf 初始化器的值为 true，或者本地系统中没有哪个 Actor 对象的路径匹配指定的 path 初始化器，message 初始化器只会被发送给其他 Actor 系统中的 Actor 对象</p>
DistributedPubSubMediator. Publish	<p>如果你想要向集群中所有订阅了某个主题的 Actor 对象广播消息，可使用 Publish 消息</p> <p>消息样本类 Publish 接收两个初始化器：topic 和 message。message 初始化器是最终会被广播的消息，而且会被发送给集群中所有符合 topic 初始化器的 Actor 对象。topic 初始化器与 path 初始化器不同，它不是用于标识指定 Actor 对象的，而且 topic 初始化器的名称应反映出它代表的主题。与前面介绍的使用 /user/bidProcessor 路径标识接收者 Actor 对象不同，主题的名称更适合使用 bids</p> <p>通过使用 Subscribe 消息，订阅者可以在中间人 Actor 对象中注册自己。Subscribe 消息接收两个初始化器：topic 和 actorRef。topic 初始化器中含有字符串类型的广播消息接收者 Actor 对象的名称，而 actorRef 初始化器中含有订阅者 Actor 对象的引用。当订阅者 Actor 对象收到 SubscribeAck 消息时，就表明该对象已经被成功注册。SubscribeAck 消息中含有原始的 Subscribe 消息</p> <p>当 Publish 消息被发送给中间人 Actor 对象时，所有订阅了 topic 初始化器中包含的主题的订阅者 Actor 对象都会被发送 message 初始化器</p> <p>订阅者 Actor 对象可以通过向中间人 Actor 对象发送 Unsubscribe 消息，将自己从该中间人 Actor 对象的已注册订阅者名单中移除。当订阅者 Actor 对象收到 UnsubscribeAck 消息后，就表明该订阅者 Actor 对象已经成功将自己从中间人 Actor 对象的已注册订阅者名单中移除。UnsubscribeAck 消息含有原始的 Unsubscribe 消息</p>

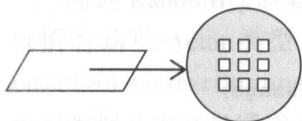
可以使用两种方式向本地中间人 Actor 对象的订阅者名单中添加条目, 和从该名单中删除条目。这样就使这些条目代表的 Actor 对象能够接收 DistributedPubSubMediator 对象发布的消息。

- 当消息是由 Send 或 SendToAll 方法 (请参阅表 5.3) 发送时, 想要接收该消息的 Actor 对象就必须向它们的本地中间人 Actor 对象发送一条 DistributedPubSubMediator.Put 消息, 以便将自己添加到本地中间人 Actor 对象的订阅者名单中。这条 DistributedPubSubMediator.Put 消息中的唯一参数, 是想要订阅消息的 Actor 对象的 ActorRef 引用。此处的要点是, 对于本地中间人 Actor 对象来说, 所有想要订阅消息的 Actor 对象都位于本地。换言之, 中间人 Actor 对象和这些想要订阅消息的 Actor 对象都处于同一个 ActorSystem 系统中。要从本地中间人 Actor 对象的订阅者名单中删除已注册的 Actor 对象, 可使用 DistributedPubSubMediator.Remove 消息。该消息中含有的唯一参数, 是要删除的 Actor 对象的路径 (String 类型的)。
- 当消息是由 Publish 方法 (请参阅表 5.3) 发送时, 想要接收该消息的 Actor 对象就必须向它们的本地中间人 Actor 对象发送一条 DistributedPubSubMediator.Subscribe 消息, 以便将自己添加到本地中间人 Actor 对象的订阅者名单中。这条 DistributedPubSubMediator.Subscribe 消息中含有两个参数。第一个参数是 String 类型的订阅主题, 第二个参数是想要订阅消息的 Actor 对象的 ActorRef 引用。成功将想要订阅消息的 Actor 对象添加到本地中间人 Actor 对象的订阅者名单中后, 本地中间人 Actor 对象会向想要订阅消息的 Actor 对象发送一条 DistributedPubSubMediator.SubscribeAck 消息。同样, 此处的要点还是对于本地中间人 Actor 对象来说, 所有想要订阅消息的 Actor 对象都位于本地。换言之, 中间人 Actor 对象和这些想要订阅消息的 Actor 对象都处于同一个 ActorSystem 系统中。要从本地中间人 Actor 对象的订阅者名单中删除已注册的 Actor 对象, 可使用 DistributedPubSubMediator.Unsubscribe 消息。这条消息中含有两个参数, 一个参数是 String 类型的订阅主题, 另一个参数是要从本地中间人 Actor 对象的订阅者名单中删除的已注册 Actor 对象的 ActorRef 引用。成功从本地中间人 Actor 对象的订阅者名单中删除已注册的 Actor 对象后, 取消订阅的 Actor 对象会从本地中间人 Actor 对象那里收到回复消息 DistributedPubSubMediator.UnsubscribeAck。

此外，当 Actor 对象停止运行后，会自动将自己从本地 Distributed-PubSubMediator 对象的订阅者名单中删除，即取消对本地 DistributedPubSubMediator 对象所发布的信息进行订阅。

掌握了上述知识后，你就可以在 Akka 集群的分布式节点中使用发布—订阅通道了。

数据类型通道



当接收者应用程序需要在不检验消息内容的情况下，就必须了解收到消息的数据类型时，可使用数据类型通道。可根据不同的数据类型通道，创建不同类型的 Actor 对象。数据类型通道既具有消息通道的特点，又具有消息端点的特点。图 5.4 展示了数据类型通道的运行方式。

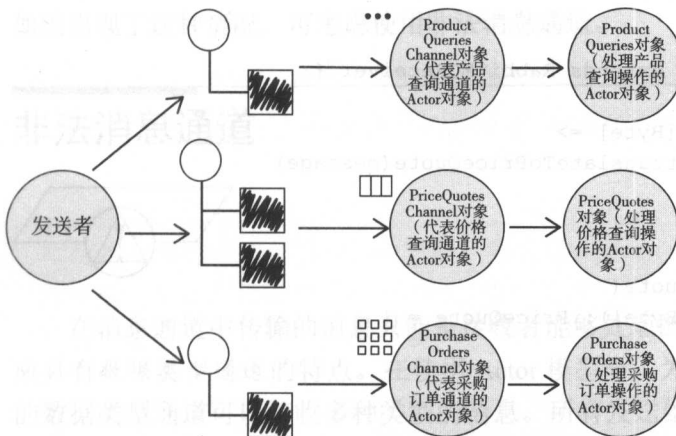


图 5.4 在使用数据类型通道时，发送者需要根据消息的数据类型，将它们发送给能够处理相应数据类型消息的 Actor 对象。

在使用 Actor 模型时，通常需要为专门处理某一数据类型的数据类型 Actor 对象，定义具有类型安全性的消息。每种消息都代表一种数据类型。即使某个 Actor 对象能够接收多种类型的消息，但只要明确了可接收消息类型的协定，该 Actor 对象的消息通道就成为数据类型通道。

即使如此，有时可能无法使用不同类型的消息代表不同的数据类型。当

Actor 对象从中间件服务器接收消息时就会出现这种情况。例如，你可能需要将 Actor 对象定义为接收来自 RabbitMQ 消息代理系统或 JMS 接口的消息，这类消息都是由二进制字节数组构成的。²

在下面的示例代码中，每个 Actor 对象都扩展了一个专用的抽象 Actor 类型——RabbitMQReceiver。这意味着所有 Actor 对象都是由监听已命名通道的 RabbitMQ Java 客户端支持的。

// 监听产品查询通道

```
class ProductQueriesChannel extends RabbitMQReceiver {
  def receive = {
    case message: Array[Byte] =>
      val productQuery = translateToProductQuery(message)
      ...
  }

  def translateToProductQuery(
    message: Array[Byte]): ProductQuery = {
    ...
  }
}
```

// 监听报价通道

```
class PriceQuoteChannel extends RabbitMQReceiver {
  def receive = {
    case message: Array[Byte] =>
      val priceQuote = translateToPriceQuote(message)
      ...
  }

  def translateToPriceQuote(
    message: Array[Byte]): PriceQuote = {
    ...
  }
}
```

// 监听采购订单通道

```
class PurchaseOrderChannel extends RabbitMQReceiver {
  def receive = {
    case message: Array[Byte] =>
```

² 这类消息除了能由二进制数构成，还可以由文本（如XML）构成，这取决于消息的结构。而且这类消息还可以是JSON格式的，并且需要符合弱架构的要求。综上所述，二进制格式仅是这些可用格式的一个例子。

```

val purchaseOrder = translateToPurchaseOrder(message)
...
}

def translateToPurchaseOrder(
  message: Array[Byte]): PurchaseOrder = {
  ...
}
}

```

因为 RabbitMQ 消息代理系统不支持具有类型安全性的消息交换操作，所以所有这些 Actor 类型（ProductQueriesChannel、PriceQuoteChannel 和 PurchaseOrderChannel）都接收全部由字节数组构成的消息。因此，消息的数据类型是由消息通道决定的。这些 Actor 对象都使用消息译码器，将收到的字节数组转换为指定数据类型的实例，以便能够成功地处理这些消息。

通过实现抽象基类 RabbitMQReceiver，可以执行这些转换操作，从而能够将具有类型安全性的消息交给每个 Actor 对象。这个设计思路可能很有价值。然而，前面的示例已经展示了处理特定类型通道传输的消息所需的必要步骤。

还可能出现消息发送者将非法类型的消息误传给指定数据类型通道的情况。如果出现了这种情况，可考虑使用非法消息通道。

非法消息通道



在消息通道中传输的消息只能是接收者能够处理的消息。换言之，消息通道应具有数据类型通道的特点。在使用 Actor 模型时，为指定 Actor 类型传输消息的数据类型通道可以接收多种类型的消息。所有发送给 Actor 对象的消息都必须遵守 Actor 对象的协定。你可以使用非法消息通道（如图 5.5 所示），处理不符合协定的消息。

死信通道通常用于为无法送达的消息提供路由，而非法消息通道用于为接收者无法处理的消息提供路由。即便如此，在使用 Akka 框架时，也可以将非法消息发送给死信通道。

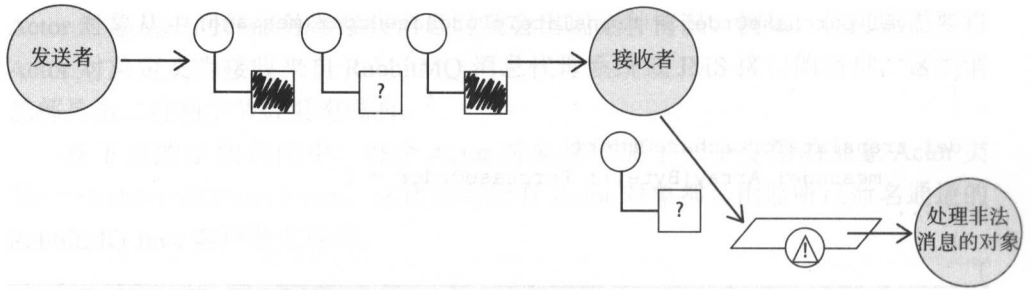


图 5.5 应将接收者无法处理的消息发送给非法消息通道。

如果你选择创建独立的非法消息通道，使用 Akka 框架可以轻松达到目的。

```

case class InvalidMessage(
  sender: ActorRef,
  receiver: ActorRef,
  message: Any)

class InvalidMessageChannel extends Actor {
  def receive = {
    case invalid: InvalidMessage =>
      ...
  }
}
  
```

请回顾管道和过滤器的一些示例，并思考当收到非 `ProcessIncomingOrder` 消息时，`Authenticator` 对象应该执行哪些操作。

```

class Authenticator(
  nextFilter: ActorRef,
  invalidMessageChannel: ActorRef)
  extends Actor {
  def receive = {
    case message: ProcessIncomingOrder =>
      val text = new String(message.orderInfo)
      ...
      nextFilter ! ProcessIncomingOrder(
        orderText.toCharArray.map(_.toByte))
    case invalid: Any =>
      invalidMessageChannel ! InvalidMessage(
        sender, self, invalid)
  }
}
  
```

此处的要点是，一旦收到非法消息，InvalidMessageChannel 对象应该执行哪些操作？是否应该将该消息记录到应用程序的错误日志中？是否应该通知原始发送者？是否应该将该消息存储到永久存储设备中稍后再进行处理，或者在原接收者被重新部署因而能够处理先前的非法消息后重新向它发送该消息？非法消息应该支持所有这些可能出现的情况。

然而，如果应用程序除了将非法消息记入日志外不执行其他处理操作，原接收者（本例中为 Authenticator 对象）可以执行下列操作。

```
import akka.actor._
import akka.event.Logging
import akka.event.LoggingAdapter

class Authenticator(nextFilter: ActorRef) extends Actor {
  private val log: LoggingAdapter =
    Logging.getLogger(context.system, self)
  def receive = {
    case message: ProcessIncomingOrder =>
      val text = new String(message.orderInfo)
      ...
      nextFilter ! ProcessIncomingOrder(
        orderText.toCharArray.map(_._toByte))
    case invalid: Any =>
      log.error(s"Unknown: ${invalid.getClass.getName}")
  }
}
```

在这段代码中，LoggingAdapter 事件代表非法消息通道。尽管 LoggingAdapter 事件本身不是 Actor 对象，但它是由 Actor 系统的 EventBus 接口支持的并且以异步方式运行。EventBus 接口实现了一个通用的发布—订阅通道，该通道既能在系统层面使用也能在应用程序层面使用。

死信通道



当消息系统无法将消息送达接收者时，可以选择将消息发送给死信通道。实际上这也是 Akka 框架处理死信消息的方式，如图 5.6 所示。

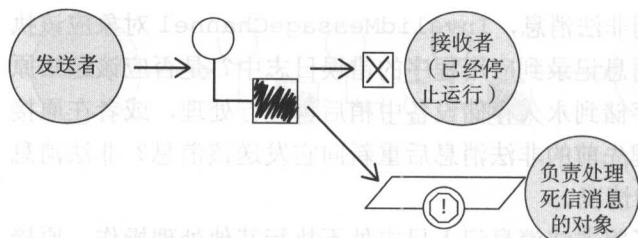


图 5.6 当无法将消息送达接收者时，可以通过死信通道处理消息。

在使用 Actor 模型（特别是使用 Akka 框架）时，可使用基于 Actor 的死信通道，名为 `/deadLetters`。每个 Actor 系统都会创建这种特殊的 Actor 对象，而且当 Akka 确定无法将某一条消息送达接收者时，就会将这条消息发送给名为 `/deadLetters` 的 Actor 对象。

下面是 Akka 框架使用死信通道的原因。

- 当一个本地 Actor 对象向另一个本地 Actor 对象发送消息时，可能会出现发送消息时接收者 Actor 对象已经不存在的情况。出现这种情况时，Akka 框架会将该消息发送给本地的 Actor 对象 `deadLetters`。然而，你可能会好奇，为什么会出现向不存在的接收者发送消息的情况呢？答案很简单，因为 `ActorRef` 引用和 Actor 对象的实现代码之间的耦合性被降得很低。因为发送者仅是通过 `ActorRef` 引用接收者 Actor 对象，所以在发送者向接收者发送消息时，接收者有可能正在停止运行。当 Akka 框架检测到这个情况时，就会将无法送达的消息发送给本地系统中的 `deadLetters` 对象。
- 当一个节点中的 Actor 对象向另一个节点中的 Actor 对象发送消息时，两个节点之间的网络可能会出故障。当 Akka 框架在发送者节点上检测到这个情况时，就会将无法送达的消息发送给发送者节点中的 `deadLetters` 对象。
- 当一个节点中的 Actor 对象向另一个节点中的 Actor 对象发送消息时，两个节点之间的网络运行正常，但在发送消息时远程节点中的接收者 Actor 对象已经不存在了。当 Akka 框架检测到这个情况时，就会将无法送达的消息发送给远程系统（即先前存在的接收者 Actor 对象所在的）中的 `deadLetters` 对象。

那么基于 Akka 的应用程序怎样才能了解到消息已经被传送给 `deadLetters` 对象了呢？要做到这一点，只需通过本地 Actor 系统中的 `EventStream` 对象，注册一个起监听器作用的 Actor 对象。

```
val system = ActorSystem("TradingSystem")
val sysListener = system.actorOf(Props[SystemListener], "sysListener")
system.eventStream.subscribe(
    sysListener, classOf[akka.actor.DeadLetter])
```

注册之后，通过 `sysListener` 变量引用的 `Actor` 对象，会从 `Actor` 系统的 `EventStream` 对象那里接收到 `akka.actor.DeadLetter` 类型的消息。

```
class SystemListener extends Actor {
  def receive = {
    case deadLetter: DeadLetter =>
      ...
  }
}
```

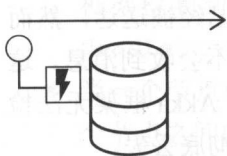
你现在可能已经能够推测出，使用 `DeadLetter` 监听器实现确保送达通道的思路。因为 `DeadLetter` 消息既含有发送者的 `ActorRef` 引用，也含有接收者的 `ActorRef` 引用，你可能会认为 `SystemListener` 对象只需通过动态方式重新加载接收者，并重新向其发送消息。当网络出现故障时，你可能也会认为可以使用类似的方式重新连接远程节点。然而，在继续思考这些推论前，请看看下面通过 `deadLetters` 对象为无法送达的消息提供路由的方式。

- Akka 框架有可能无法检测到消息接收者正在停止运行。当出现这种情况时，Akka 框架会尝试将消息传送给接收者并假定消息已经被送达，然而实际上正在停止和已经停止运行的 `Actor` 对象，永远都不会收到消息。这通常会导致在发送者和接收者线程之间出现竞态条件，Akka 框架无法检测到这个问题。当出现这种情况时，无法送达的消息会彻底丢失。
- 本地 Akka 系统可以通过网络向远程节点中的 `Actor` 对象发送消息，但是在传输消息的过程中网络可能会出现故障。本地 Akka 系统可能会误认为网络传输操作已经成功完成，但事实上消息没有成功送到远程节点。根据 Akka 框架的实现方式，接收端和发送端的 TCP 网络连接都不会知道消息未被送达，因此未送达的消息既不会被发送给发送端的 `deadLetters` 对象，也不会被发送给接收端的 `deadLetters` 对象。

这些情况不是 Akka 框架部分的故障。更确切地说，这是 `Actor` 模型固有的问题：消息至多仅会被发送一次。对于许多应用程序来说，至多发送一次的传输方式是完全可以接受的，因为这能够提高吞吐量并降低确保送达机制带来的不必要的系统开销。

这些结论不意味着 Akka 框架无法支持确保将消息送达的高服务等级协议。请参阅前面介绍消息通道和确保送达通道的内容，以便详细了解这方面的知识。然而，如果出现了前面介绍的情况，就无法确保使 Akka 框架的死信通道支持确保送达功能，因为这不是一种完全可靠的处理方式。Akka 框架中的 `deadLetters` 对象不是这样用的。更确切地说，该对象更像一种调试工具，它使开发者能够在应用程序中使用某种消息传输设计策略时，通过监视死信消息的传送情况，查明实现代码中的问题。前面介绍过，根据导致消息无法被送达的原因，向远程节点中 Actor 对象发送的消息，可能接受本地节点中 `EventStream` 对象提供的路由，也可能接受远程节点中 `EventStream` 对象提供的路由。因此，你不得不观察两个或多个 `EventStream` 对象的反馈信息，才能查明某条未送达消息传输失败的原因。是否能够通过某种方式，将多个节点的 `EventStream` 反馈信息整合到一起呢？Akka 框架没有提供这个功能，但你可以自己编写这个功能。实现这种集成式的 `EventStream` 对象，需要在应用程序多个节点其中之一上实现一个聚合器。如果你使用了 Akka 集群，就应该像部署集群单例对象一样部署这个聚合器（请参阅 Akka 文档对 `ClusterSingletonManager` 单例对象的介绍）。然而，如果你整合了不会参与该聚合操作的第三方系统，那么这种处理方式的难度要大得多，而且有时可能无法实现。

确保送达机制



当需要确保将指定的消息送达接收者时，应使用确保送达机制。通过 Akka 框架使用这种模式，可以确保至少将消息向其接收者发送一次。要做到这一点，需要将消息存储到消息存储器中，然后定期发送该消息直到收到用于确定该消息被送达的回执（也会被存储在消息存储器中）为止。除非特意删除某条消息，否则所有消息都会永久地被保存。

通常第一次发送消息后，发送者就会收到确认消息已被送达的消息。收到确认后，Akka 框架会防止再次发送相同的消息。然而，如果在一个发送间隔中第一次发送的消息没有被送达，那么该消息至少还会被再发送一次。因为各种延迟因素和丢失确认回执的情况，这种重复发送消息的方式可能会使接收者，两次或多次收到同一条消息。图 5.7 展示了 Akka 框架的确保送达协定。

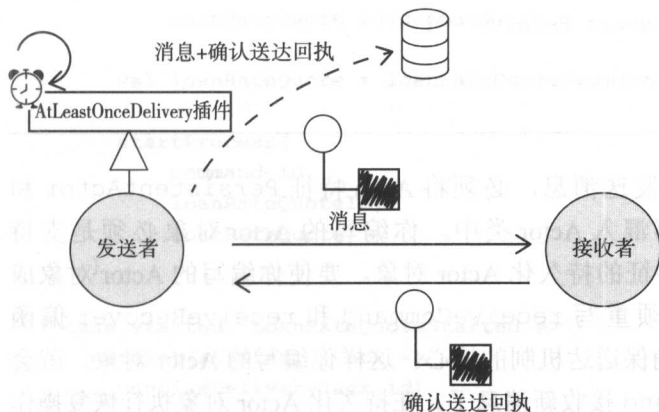


图 5.7 通过 Akka 框架的 AtLeastOnceDelivery 插件遵守确保送达协定。

为了促进该功能，Akka 框架提供了持久化组件 Akka Persistence。要在项目中使用 Akka Persistence 组件，需要在项目的 build.sbt 文件中添加下面的 JAR 文件。

- akka-persistence_x.y.z.jar : Akka Persistence 组件。在撰写本书时，该组件仍旧处于试验阶段，它当时的文件名为 akka-persistence-experimental_2.10-2.3.6.jar。
- leveldb-x.y.jar : LevelDB 数据库功能。
- leveldb-api-x.y.jar : LevelDB 应用程序编程接口（API）。
- leveldbjni-all-x.y.jar : LevelDB Java 本地接口（JNI）链接。
- protobuf-java-x.y.z.jar : Java 版本的 Google ProtoBuf 库。

上面的文件名中都含有 x.y.z，它们代表版本号。尽管 LevelDB 是默认的存储格式，但并不是说产品级的消息存储器必须使用这种数据库。使用各种第三方软件也可以完成这项数据存储任务。然而，上面介绍的 JAR 文件能够帮助你更好更快地使用 Akka Persistence 组件。

当配置好编写和运行代码的环境后，就可以创建支持确保送达机制的 Actor 对象了。

```

class LoanBroker
  extends PersistentActor
  with AtLeastOnceDelivery {

  override def receiveCommand: Receive = {
    ...
  }
}
    
```



```

override def receiveRecover: Receive = {
  ...
}
}

```

要通过确保送达机制发送消息，必须将 Akka 特征 `PersistentActor` 和 `AtLeastOnceDelivery` 都混入 `Actor` 类中。你编写的 `Actor` 对象必须是支持 `AtLeastOnceDelivery` 特征的持久化 `Actor` 对象。要使你编写的 `Actor` 对象成为持久化 `Actor` 对象，必须重写 `receiveCommand` 和 `receiveRecover` 偏函数，在这些偏函数中添加确保送达机制的核心。这样你编写的 `Actor` 对象，就会通过偏函数 `receiveCommand` 接收新消息了。在持久化 `Actor` 对象执行恢复操作时，会处理所有之前持久化（即把数据，如内存中的对象，保存到可永久保存的存储设备中，如磁盘）的事件消息，并通过偏函数 `receiveRecover` 发送确认收到回执。请参阅第 9 章，详细了解 `PersistentActor` 特征。下面将详细介绍 `AtLeastOnceDelivery` 特征。

让我们先看看使用确保送达机制时发送消息的必要步骤。下面是 `LoanBroker` 类中的 `receiveCommand` 偏函数。

```

class LoanBroker
  extends PersistentActor
  with AtLeastOnceDelivery {

  private var processes =
    Map[String, LoanRateQuoteRequested]()

  ...

  def checkCredit(
    loanRateQuoteId: String,
    taxId: String) = {
    ...
  }

  def duplicate(command: QuoteBestLoanRate): Boolean = {
    ...
    if (duplicateFound) {
      sender ! loanRateQuoteRequested
    }
  }
  ...

  def receiveCommand: Receive = {
    case command: QuoteBestLoanRate =>
      if (!duplicate(command)) {
        val loanRateQuoteId =

```

```

LoanRateQuote.id(command.id)

val loanRateQuote = loanRateQuoteFrom(command)

startProcess(
  command.id,
  loanRateQuoteId,
  loanRateQuote)
}

case started: LoanRateQuoteStarted =>
  persist(started) { ack =>
    confirmDelivery(ack.id)
  }
  checkCredit(started.loanRateQuoteId, started.taxId)
  ...
}

override def receiveRecover: Receive = {
  ...
}

def startProcess(
  quoteRequestId: Int,
  loanRateQuoteId: String,
  loanRateQuote: ActorRef) = {

  val loanRateQuotePath = loanRateQuote.path

  val loanRateQuoteRequested =
    LoanRateQuoteRequested(
      quoteRequestId,
      loanRateQuoteId,
      loanRateQuotePath)

  persist(loanRateQuoteRequested) { event =>
    updateWith(event)

    deliver(
      loanRateQuotePath,
      id => StartLoanRateQuote(totalBanks, id))
    sender ! event
  }
}

def totalBanks(): Int = {
  ...
}

```

```

}
...
def updateWith(event: LoanRateQuoteRequested) = {
  if (!processes.contains(event.processId)) {
    processes = processes +
      (event.processId -> event)
  }
}
}

```

在偏函数 `receiveCommand` 中，应先处理命令消息 `QuoteBestLoanRate`。收到该消息后，应先确保该命令消息不是被重复发送的副本消息。此处的协定是客户端 `Actor` 对象，还应拥有 `AtLeastOnceDelivery` 特征。如果这条消息是被重复发送的消息副本，就说明最初发送给客户端的事件消息已经作为确认收到回执被保存了起来，此时的最佳选择是向客户端重新发送这个确认收到回执，这样客户端就不会反复发送命令消息 `QuoteBestLoanRate` 了。`startProcess()` 方法中含有事件消息 `LoanRateQuoteRequested` 的起源代码，而该消息会作为运行状态被 `updateWith()` 方法保存起来。如果该消息不是被重复发送的消息副本，你就可以创建一个新的 `loanRateQuoteId` 对象和相关的 `loanRateQuote` 对象，然后调用 `startProcess()` 方法。³

在 `startProcess()` 方法中，应通过 `loanRateQuotePath` 变量引用 `Actor` 对象 `LoanRateQuote` 的 `ActorPath` 字段。然后创建 `LoanRateQuoteRequested` 类型的新事件消息，并将该消息存储到消息存储器中（在创建和保存 `LoanRateQuoteRequested` 消息前，可能还需要验证收到的命令或 `Actor` 对象的当前状态，本例省略了该步骤）。

`persist()` 方法的输出结果会使相关的处理程序代码块被执行。该处理程序的第一个步骤是使用 `updateWith()` 方法更新 `LoanBroker` 对象的运行状态，并传递 `event` 参数。简言之，`event` 参数是与 `LoanRateQuoteRequested` 相同的实例，是被保存起来的运行状态，用于检验消息是否为重复发送的消息副本。当处理命令消息的过程完成后，该状态就能够被安全地移除。

这种确保送达模式的要点是，可以使用专用的 `AtLeastOnceDelivery` 操作将命令消息 `StartLoanRateQuote` 发送给 `Actor` 对象 `LoanRateQuote`。要做到这一点，可使用 `deliver()` 代码块，传送 `Actor` 对象 `LoanRateQuote` 的 `ActorPath` 字段和命令消息 `StartLoanRateQuote`。前面介绍过使用 `Actor` 路

³ 要详细了解该处理过程的整体运行方式，请参阅第7章介绍的处理过程管理器示例。

径和 `actorSelection()` 方法会使发送消息的操作更加可靠，因为这可以找出相关 `Actor` 对象大部分最近生成的实例。

`deliver()` 代码块的第二个参数看起来有点奇怪。这段代码将一个函数参数传递给 `deliver()` 代码块。这个函数参数含有用于创建 `StartLoanRateQuote` 实例的代码，但是需要由 `deliver()` 代码块提供 `id` 参数才能完整地创建该实例。`deliver()` 代码块在其内部通过私有方法 `nextDeliverySequenceNr()` 生成了新的 `id`，然后将它传送给函数参数。该 `id` 的名称为 `deliveryId`，当该 `id` 随 `LoanRateQuoteStarted` 确认送达消息被回复后，该 `id` 一定会被 `Actor` 对象 `LoanRateQuote` 使用。

`persist()` 代码块的最后一个语句，使用事件消息 `LoanRateQuoteRequested` 对命令消息 `QuoteBestLoanRate` 的发送者做出了回复。如果发送 `QuoteBestLoanRate` 消息（被 `receiveCommand` 偏函数接收的）的客户端，本身是混入 `AtLeastOnceDelivery` 特征的持久化 `Actor` 对象，那么通过 `QuoteBestLoanRate` 消息中的 `deliveryId` 字段初始化的 `LoanRateQuoteRequested` 对象，就会起到确认回执的作用，用于表明已经收到客户端的请求。当客户端收到 `LoanRateQuoteRequested` 消息后，`LoanRateQuoteRequested` 对象会阻止 `LoanBroker` 对象接收重复发送的 `QuoteBestLoanRate` 消息副本。

最后让我们看看偏函数 `receiveCommand`，请注意匹配的是 `LoanRateQuoteStarted` 消息。当 `LoanBroker` 对象处理了命令消息 `StartLoanRateQuote` 后，`Actor` 对象 `LoanRateQuote` 就会发送事件消息 `LoanRateQuoteStarted`。`LoanBroker` 对象必须既保存 `LoanRateQuoteStarted` 消息，又使用该消息调用 `confirmDelivery()` 代码块。此处会再次用到 `LoanRateQuoteStarted` 对象的 `deliveryId` 字段，该字段与 `deliver()` 代码块第一次创建 `StartLoanRateQuote` 对象时，在 `startProcess()` 方法内部生成的 `deliveryId` 字段相同（值 `started` 和 `ack` 引用了同一个 `LoanRateQuoteStarted` 消息对象）。调用了 `confirmDelivery()` 代码块后，`Actor` 对象 `LoanRateQuote` 永远不会再收到含有该 `deliveryId` 字段的 `StartLoanRateQuote` 消息。偏函数 `receiveRecover` 用于通过已存储的事件重新转换 `LoanBroker` 对象的状态。而且，`receiveRecover` 偏函数还有处理 `AtLeastOnceDelivery` 特征的作用：

```
class LoanBroker
  extends PersistentActor
  with AtLeastOnceDelivery {
  ...
```

```

override def receiveRecover: Receive = {
  case event: LoanRateQuoteRequested =>
    updateWith(event)

    deliver(
      event.loanRateQuotePath,
      id => StartLoanRateQuote(totalBanks, id))

  case started: LoanRateQuoteStarted =>
    confirmDelivery(started.id)
  ...
}
...
}

```

因为你可以重组 LoanBroker 对象的状态，所以该对象可以处理任何已存储的 LoanRateQuoteRequested 事件。像使用 persist() 处理程序代码块一样，还可以使用 updateWith() 方法更新 LoanBroker 对象的状态和传递 LoanRateQuoteRequested 实例。之后，还可以尝试传送相关的 StartLoanRateQuote 命令消息。这能够确保当 LoanBroker 对象停止运行后，所有未发送的消息会再一次被添加到至少发送一次的处理过程中。

如果与指定的 LoanRateQuoteRequested 事件关联的特殊 StartLoanRateQuote 消息已经被发送了，会出现怎样的情况呢？这个被重复执行的 deliver() 代码块是否会导致不必要的结果，并且因此而不断重复地发送该消息呢？实际上，这是恢复模式的关键和第二种需要处理的情况。

前面介绍过，LoanBroker 对象会保存来自 Actor 对象 LoadRateQuote 的 LoanRateQuoteStarted 消息。当 LoanBroker 对象处理已保存的消息时，会处理所有 LoanRateQuoteStarted 消息。在执行这些操作时，会使用 confirmDelivery() 代码块并传递该消息的 deliveryId 字段。在使用恢复模式时，实际上发送给 deliver() 代码块的消息都不会被发送。但是，AtLeastOnceDelivery 插件会等待直到恢复处理过程彻底执行完为止，以便检查仍旧没有被 confirmDelivery() 代码块确认送达的消息。只有这些没有被确认送达的消息 (deliveryId 字段没有被 confirmDelivery() 代码块确认，因而被传递给 deliver() 代码块)，才会被重新发送。

图 5.8 展示了 LoanBroker 和 LoanRateQuote 对象完整的至少发送一次的处理过程。

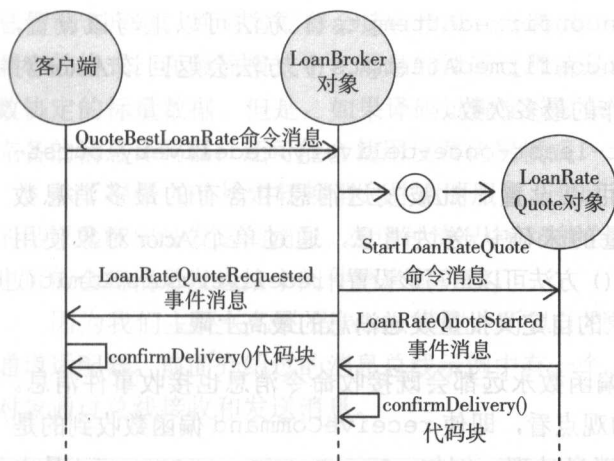


图 5.8 客户端、LoanBroker 和 LoanRateQuote 对象之间的交互操作，表现为消息序列和这些消息序列出现时被调用的方法。

你可以设置通过 `AtLeastOnceDelivery` 操作重新发送未确认送达消息的时间间隔。下面是具体的配置和重写代码。

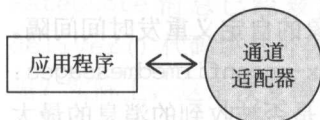
- `akka.persistence.at-least-once-delivery.redeliverinterval`: 使用这段代码可以设置执行重新发送操作的时间间隔。通过单个 Actor 对象使用 `redeliverInterval()` 方法可以重写该设置，`redeliverInterval()` 方法会返回该 Actor 对象的自定义重发时间间隔。
- `akka.persistence.at-least-once-delivery.max-unconfirmedmessages`: 使用这段代码可以设置缓存上限，以限定未确定是否被收到的消息的最大缓存值。通过单个 Actor 对象使用 `maxUnconfirmedMessages()` 方法可以重写该设置，`maxUnconfirmedMessages()` 方法会返回该 Actor 对象的自定义最大缓存值。
- `akka.persistence.at-least-once-delivery.warn-after-number-of-unconfirmed-attempts`: 执行过一定次数失败的重发操作后，当实现至少发送一次机制的 Actor 对象收到 `AtLeastOnceDelivery.UnconfirmedWarning` 消息时，这段代码可以使实现至少发送一次机制的 Actor 对象取消对指定消息执行的反复重发操作。当 Actor 对象收到 `AtLeastOnceDelivery.UnconfirmedWarning` 警告消息时，你可以使之调用 `confirmDelivery()` 代码块，从而取消重发操作。注意，因成功送达而执行的确认操作与因停止重发而执行确认操作之间没有区别，都会调用 `confirmDelivery()` 代码块。通过单个 Actor 对象使用

`warnAfterNumberOfUnconfirmedAttempts()` 方法可以重写该设置, `warnAfterNumberOfUnconfirmedAttempts()` 方法会返回该 Actor 对象的自定义执行重发操作的最多次数。

- `akka.persistence.at-least-once-delivery.redelivery-burst-limit`: 使用这段代码可以设置一批被发送消息中含有的最多消息数量, 从而防止出现大量的未确认送达消息。通过单个 Actor 对象使用 `redeliveryBurstLimit()` 方法可以重写该设置, `redeliveryBurstLimit()` 方法会返回该 Actor 对象的自定义批量发送消息的最高上限。

注意, `receiveCommand` 偏函数永远都会既接收命令消息也接收事件消息。这是因为以持久化 Actor 对象的观点看, 即使 `receiveCommand` 偏函数收到的是事件消息, 也会将之作为命令消息处理。例如, `LoanRateQuoteStarted` 是由 Actor 对象 `LoanRateQuote` 生成的事件消息, 但是对于 `LoanBroker` 对象来说, 可以将该消息视为停止执行重发操作的隐式命令消息。即使你不赞同这个观点, 仅将 `receiveCommand` 视为 Actor 对象用于接收消息的普通偏函数, 该函数也是持久化 Actor 对象用于处理所有种类消息的途径。

通道适配器



通道适配器是一种消息端点。当应用程序在系统边界提供含有集中的应用程序服务的消息传输接口时, 需要在消息传输功能和应用程序服务之间使用通道适配器。图 5.9 展示了这个情况。通道适配器可以仅由 Actor 对象构成, 如果你想系统边界既支持 Actor 模型也支持中间件消息服务器, 也可以使通道适配器既含有 Actor 对象也含有消息监听器。

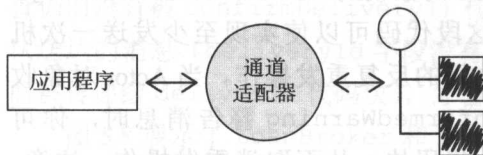


图 5.9 在应用程序服务和实际处理组件之间, 作为 Actor 对象实现的通道适配器。

在使用端口和适配器架构时 [IDDD], 通道适配器就是应用程序边界上的消

息传输客户端。通道适配器会接收消息，并将其中的有效数据转换为与内部 API 兼容的格式。在最简模式中，消息的译码过程就是将文本转换成符合 API 合法参数协定的标量数据。但是，如果译码过程比较复杂，通道适配器就需要使用功能完备的消息译码器，还可能会使用反腐化层模式。

除了根据 API 协定转换消息并调用服务，使用通道适配器还可以使应用程序的结果匹配向外发送的消息。在这种情况下，可能会将消息发送给远程 Actor 对象，也可能会将消息发送给中间件消息服务器。

因为我们主要关心的是 Actor 模型，所以让我们来观察典型的基于 Actor 的通道适配器。前面介绍过的消息总线示例中有一个 Actor 对象 StockTrader，该对象通过总线接收和发送消息。

```
class StockTrader(tradingBus: ActorRef) extends Actor {
  val applicationId = self.path.name

  tradingBus ! RegisterCommandHandler(
    applicationId,
    "ExecuteBuyOrder",
    self)
  tradingBus ! RegisterCommandHandler(
    applicationId,
    "ExecuteSellOrder",
    self)

  def receive = {
    case buy: ExecuteBuyOrder =>
      val result = buyerService.placeBuyOrder(
        buy.portfolioId,
        buy.symbol,
        buy.quantity,
        buy.price)
      tradingBus ! TradingNotification(
        "BuyOrderExecuted",
        BuyOrderExecuted(
          result.portfolioId,
          result.orderId,
          result.symbol,
          result.quantity,
          result.totalCost))

    case sell: ExecuteSellOrder =>
      val result = sellerService.placeSellOrder(
        buy.portfolioId,
        buy.symbol,
```

```

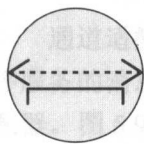
        buy.quantity,
        buy.price)
    tradingBus ! TradingNotification(
        "SellOrderExecuted",
        SellOrderExecuted(
            result.portfolioId,
            result.orderId,
            result.symbol,
            result.quantity,
            result.totalCost))
}
}

```

StockTrader 对象从消息总线接收两类消息: ExecuteBuyOrder 和 ExecuteSellOrder。作为一个通道适配器, StockTrader 对象会根据 API 的要求适配收到的消息。ExecuteBuyOrder 消息被转换为 BuyerService 对象能够接收的数据, 而 ExecuteSellOrder 消息被转换为 SellerService 对象能够接收的数据。

本例没有使用实现股票交易系统应用程序的 Actor 对象, 而是着重介绍通道适配器。这个适配器 Actor 对象既向收到的消息提供适配服务, 也向发出的消息提供适配服务。不论应用程序的内部设计使用哪些类型的组件, 这都是通道适配器的主要功能。

消息桥



在普通企业中, 会存在各种架构机制(包括各种数据库和消息系统)产生的数据负荷。如果将两个消息系统无法协同操作的应用程序整合到一起, 会出现怎样的情况呢? 是否应该使用其他方式整合它们, 如使用文件或数据库? 如果使用企业集成模式 [EIP] 能够将两个应用程序整合到一起, 那么如何在这两个应用程序各自使用自己熟悉的消息传输机制的情况下, 整合它们的消息传输功能? 使用消息桥可以解决这些问题。

如图 5.10 所示, 消息桥可以在两个独立的消息系统之间实现互通性, 使两种或多种消息传输中间件一起运行。

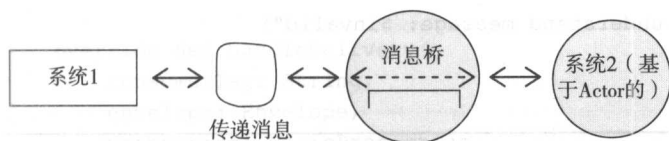


图 5.10 消息桥可以在两个独立的消息系统之间实现互通性。

如果一个应用程序使用 RabbitMQ 消息代理系统，而另一个应用程序使用 Akka 框架。你可以请使用 RabbitMQ 系统的应用程序的团队，在他们编写的应用程序中实现一些基于 Akka 的 Actor 对象。这可以确定几件事情，其中包括使用 RabbitMQ 系统的团队也会使他们开发的应用程序在 JVM 上运行。另一方面，如果该团队在 .NET 和 RabbitMQ .NET 客户端上使用 C#，该怎么办呢？本例中最不通用的元素是 RabbitMQ 系统，因为你可以通过不同的平台（如 Java 和 .NET）获得客户端。因此，我们应将关注的焦点放在 RabbitMQ-Akka 消息桥上。

另一个影响整合决策的因素是将两个应用程序整合到一起的原因。在本例中，Scala 和 Akka 应用程序是一个库存系统，.NET 应用程序是一个订单管理系统。.NET 订单管理系统需要与 Scala 和 Akka 库存系统协同工作，才能获得产品的库存信息，而且当获得一份订单时，必须向完成交易的操作链条中分配至少一件产品。如果开发 Scala 和 Akka 应用程序的团队能够编写 / 下载一个基于 Actor 的 RabbitMQ-Akka 消息桥，使 .NET 订单管理系统能够将获取库存详细信息的请求发送到 RabbitMQ 消息通道中，并通过 RabbitMQ 系统将这些请求发送给库存应用程序中的 Akka 框架 Actor 对象。这看起来像一种供应方式，因为在这种客户—供应商关系中 [IDDD]，库存应用程序扮演的是供应商的角色。

如果这就是设计目标，需要在库存应用程序中创建一个消息桥 Actor 对象。

```

class InventoryProductAllocationBridge(
    config: RabbitMQBridgeConfig)
    extends RabbitMQBridgeActor(config) {

    private val log: LoggingAdapter =
        Logging.getLogger(context.system, self)

    def receive = {
        case message: RabbitMQBinaryMessage =>
            log.error("Binary messages not supported.")
        ...
        case message: RabbitMQTextMessage =>
            log.error(s"Received text: ${message.textMessage}")
        ...
        case invalid: Any =>

```

```

        log.error(s"Don't understand message: $invalid")
        ...
    }
}

```

InventoryProductAllocationBridge 类扩展了 RabbitMQBridgeActor 基类，该基类在 RabbitMQ-Akka 消息桥中有重要作用。RabbitMQBridgeActor 类实现代码的后面，是一个起连接作用的队列通道和一个起连接 RabbitMQ 客户端作用的队列消费者。当通过该 RabbitMQ 队列通道收到消息时，该队列消费者会将该消息发送给正确的处理者。

下面是 RabbitMQBridgeActor 类的代码：

```

abstract class RabbitMQBridgeActor(
    config: RabbitMQBridgeConfig)
    extends Actor {
    private val queueChannel = new QueueChannel(self, config)

    /*-----
    def receive = {
        case message: RabbitMQBinaryMessage =>
            ...
        case message: RabbitMQTextMessage =>
            ...
        case invalid: Any =>
            ...
    }
    -----*/
}

```

显然，实际工作是由 QueueChannel 方法完成的。抽象基类 RabbitMQBridgeActor 甚至没有用于执行接收操作的代码块。执行实际操作代码块被设置为注释，以便展示具体的扩展方式。QueueChannel 方法的基础代码创建了一个 DispatchingConsumer 对象，该对象是 RabbitMQ 消息的消费者/监听器。当 DispatchingConsumer 对象 (com.rabbitmq.client.DefaultConsumer 对象) 收到调用 handleDelivery() 方法的消息时，它会重新封装该消息并将该消息发送给 Akka 框架 Actor 对象。

```

private class DispatchingConsumer(
    queueChannel: QueueChannel,
    bridge: ActorRef)
    extends DefaultConsumer(queueChannel.channel) {

```

```

override def handleDelivery(
  consumerTag: String,
  envelope: Envelope,
  properties: BasicProperties,
  body: Array[Byte]): Unit = {

  if (!queueChannel.closed) {
    handle(bridge, new Delivery(envelope, properties, body));
  }
}

override def handleShutdownSignal(
  consumerTag: String,
  signal: ShutdownSignalException): Unit = {
  queueChannel.close
}

private def handle(
  bridge: ActorRef,
  delivery: Delivery): Unit = {
  try {
    if (this.filteredMessageType(delivery)) {
      ;
    } else if (queueChannel.config.messageType == Binary) {
      bridge !
        RabbitMQBinaryMessage(
          delivery.getProperties.getType,
          delivery.getProperties.getMessageId,
          delivery.getProperties.getTimestamp,
          delivery.getBody,
          delivery.getEnvelope.getDeliveryTag,
          delivery.getEnvelope.isRedeliver)
    } else if (queueChannel.config.messageType == Text) {
      bridge !
        RabbitMQTextMessage(
          delivery.getProperties.getType,
          delivery.getProperties.getMessageId,
          delivery.getProperties.getTimestamp,
          new String(delivery.getBody),
          delivery.getEnvelope.getDeliveryTag,
          delivery.getEnvelope.isRedeliver)
    }
  }
}
...

```

具体的消息桥 Actor 对象可以接收二进制 (RabbitMQBinaryMessage) 或

文本 (RabbitMQTextMessage) 消息。下面是这些类型的消息和几个支持它们的对象：

```
object RabbitMQMessageType extends Enumeration {
  type RabbitMQMessageType = Value
  val Binary, Text = Value
}

import RabbitMQMessageType._

case class RabbitMQBridgeConfig(
  messageTypes: Array[String],
  settings: RabbitMQConnectionSettings,
  name: String,
  messageType: RabbitMQMessageType,
  durable: Boolean,
  exclusive: Boolean,
  autoAcknowledged: Boolean,
  autoDeleted: Boolean) {

  if (messageTypes == null)
    throw new IllegalArgumentException(
      "Must provide empty messageTypes.")
  if (settings == null)
    throw new IllegalArgumentException(
      "Must provide settings.")
  if (name == null || name.isEmpty)
    throw new IllegalArgumentException(
      "Must provide name.")
}

case class RabbitMQConnectionSettings(
  hostname: String,
  port: Int,
  virtualHost: String,
  username: String,
  password: String) {

  def this() =
    this("localhost", -1, "/", null, null)

  def this(hostname: String, virtualHost: String) =
    this(hostname, -1, virtualHost, null, null)

  def hasPort(): Boolean =
    port > 0
}
```

```

def hasUserCredentials(): Boolean =
  username != null && password != null
}

case class RabbitMQBinaryMessage(
  messageType: String,
  messageId: String,
  timestamp: Date,
  binaryMessage: Array[Byte],
  deliveryTag: Long,
  redelivery: Boolean)

case class RabbitMQTextMessage(
  messageType: String,
  messageId: String,
  timestamp: Date,
  textMessage: String,
  deliveryTag: Long,
  redelivery: Boolean)

```

这段代码中包含了一个枚举型的 `RabbitMQMessageType` 对象，用于为指定的消息桥确定消息的类型（二进制或文本）。`RabbitMQBridgeConfig` 对象用于声明消息桥 Actor 对象的完整配置。表 5.4 介绍了所有配置。

表 5.4 消息桥 Actor 对象的 `RabbitMQBridgeConfig` 配置

参数	类型和描述
<code>messageTypes</code>	该参数的类型为 <code>Array[String]</code> ，其中含有消息桥能够接收的所有消息类型的名称。如果消息桥能够接收任何类型的消息，那么该参数就会为空（但是不能为 <code>null</code> ）
<code>settings</code>	用于设置与 <code>RabbitMQ</code> 系统的连接方式
<code>name</code>	用于设置被监听可接收消息类型队列的字符串类型的名称
<code>messageType</code>	用于设置 <code>RabbitMQ</code> 消息的类型，只能使用两个值： <code>Binary</code> 或 <code>Text</code>
<code>durable</code>	该参数为布尔型，用于设置可接收消息类型队列的长度是否可变
<code>exclusive</code>	该参数为布尔型，用于设置队列是否能够被独占
<code>autoAcknowledged</code>	该参数为布尔型，用于设置是否能够自动识别收到的消息
<code>autoDeleted</code>	该参数为布尔型，用于设置在封装时，是否能够自动删除队列

`InventoryProductAllocationBridge` 对象被配置为接收文本（`Text`）消息（而不是二进制（`Binary`）消息）。因此，消费者和供应商都必须同意使用文本格式的消息。通常，使用弱模式的 JavaScript 对象表示法（JSON）（而不是严格的可扩展标记语言 XML 格式），可以获得最佳的整合支持。如果

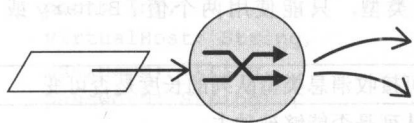
InventoryProductAllocationBridge 对象接收 JSON 格式的消息，那么它用于处理接收消息的代码块就会使用消息译码器，将有效的 JSON 文本转换为本地应用程序能够接收的数据类型。

```
class InventoryProductAllocationBridge(
  config: RabbitMQBridgeConfig)
  extends RabbitMQBridgeActor(config) {

  private val log: LoggingAdapter =
    Logging.getLogger(context.system, self)
  def receive = {
    case message: RabbitMQTextMessage =>
      val inventoryProductAllocation =
        translateToInventoryProductAllocation(
          message.textMessage)
    ...
    acknowledgeDelivery(message)
  }
}
```

一旦 RabbitMQTextMessage 消息被 acknowledgeDelivery() 方法处理完，消息桥 Actor 对象就会了解该消息已经被处理好。如果 InventoryProductAllocationBridge 对象通过异步方式将已转换的 InventoryProductAllocation 消息分派给另一个 Actor 对象进行处理，那么它就需要在另一个 Actor 对象成功做出回复后，才能使用 acknowledgeDelivery() 方法。

消息总线



企业中含有许多独立的业务系统，这些业务系统包括从购买日用品的应用程序，到为了提高竞争力用于帮助整合商务伙伴的自定义应用程序的各种系统。尽管这些系统在不同的平台上运行并且拥有各种服务接口，而且每组服务接口都有专用的数据模型，但你仍然需要使所有这些系统一起协同工作。有时通过创建实现简单的面向服务架构的消息总线，可以获得最佳效果（如图 5.11 所示）。这种消息总线必须将各个已整合的应用程序中的所有服务接口联到一起，并且使这些

服务接口都使用相同的规范化消息模型。

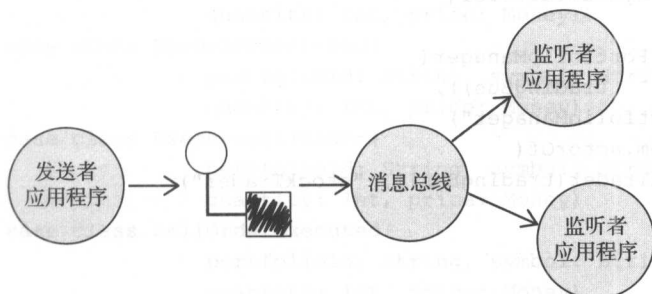


图 5.11 将各个已整合的应用程序中的所有服务接口联合到一起的消息总线。

下面的例子介绍了一个含有 3 个子系统 (Stock Trader、Portfolio Manager 和 Market Analysis Tools) 的股票交易系统。我创建了一个名为 TradingBus 的 Actor 对象, 该对象被用作消息总线。我还为每个子系统添加了用作连接器的 Actor 对象: StockTrader、PortfolioManager 和 MarketAnalysisTools。

这些系统很可能都需要使用通道适配器和消息桥, 才能通过 TradingBus 对象彻底整合到一起。通道适配器会接收向与总线相连的 Actor 对象 (如 StockTrader) 发送的指定消息, 并使之适配基础子系统 (如 Stock Trader) 的数据模型和服务接口。如果该基础子系统使用某种类型的消息传输机制 (如消息队列服务 (MSMQ)、JMS 实现方案、RabbitMQ 等), 消息桥就需要根据应用程序本身使用的消息传输机制创建和发送消息。当然, 通道适配器和消息桥还应该负责通过与消息总线及其规范化消息模型兼容的方式, 适配向外发送的消息/回复。

下面是示例应用程序的代码:

```

import scala.collection.mutable.Map
import akka.actor._
import co.vvaughnvernon.reactiveenterprise._

object MessageBus extends CompletableApp(9) {
  val tradingBus =
    system.actorOf(
      Props(new TradingBus(6)),
      "tradingBus")
  val marketAnalysisTools =
    system.actorOf(
      Props(
        new MarketAnalysisTools(
          tradingBus)),

```

```

        "marketAnalysisTools")
val portfolioManager = system.actorOf(
    Props(
        new PortfolioManager(
            tradingBus)),
    "portfolioManager")
val stockTrader = system.actorOf(
    Props(new StockTrader(tradingBus)), "stockTrader")

awaitCanStartNow()

tradingBus ! Status()

tradingBus ! TradingCommand(
    "ExecuteBuyOrder",
    ExecuteBuyOrder(
        "p123", "MSFT", 100, 31.85))
tradingBus ! TradingCommand(
    "ExecuteSellOrder",
    ExecuteSellOrder(
        "p456", "MSFT", 200, 31.80))
tradingBus ! TradingCommand(
    "ExecuteBuyOrder",
    ExecuteBuyOrder(
        "p789", "MSFT", 100, 31.83))

awaitCompletion
println("MessageBus: is completed.")
}

```

这个程序先创建了 TradingBus 对象，然后创建了代表子系统的 Actor 对象，并为这些对象赋予了 Actor 对象 TradingBus 的引用。TradingBus 对象负责完成下列任务：

- 管理命令处理程序和被通知者的注册名单。
- 向已注册的命令处理程序分派特定的命令。
- 向已注册的被通知者分派指定的通知。

所有与 TradingBus 对象相连的元素都应使用相同的规范化消息模型，这一点很重要。下面举了一个较小的例子，其中包含了 Actor 对象 TradingBus 的代码：

```

case class CommandHandler(
    applicationId: String,
    handler: ActorRef)

```

```

case class ExecuteBuyOrder(
    portfolioId: String, symbol: String,
    quantity: Int, price: Money)
case class BuyOrderExecuted(
    portfolioId: String, symbol: String,
    quantity: Int, price: Money)
case class ExecuteSellOrder(
    portfolioId: String, symbol: String,
    quantity: Int, price: Money)
case class SellOrderExecuted(
    portfolioId: String, symbol: String,
    quantity: Int, price: Money)
case class NotificationInterest(
    applicationId: String,
    interested: ActorRef)
case class RegisterCommandHandler(
    applicationId: String,
    commandId: String, handler: ActorRef)
case class RegisterNotificationInterest(
    applicationId: String,
    notificationId: String,
    interested: ActorRef)
case class TradingCommand(
    commandId: String, command: Any)
case class TradingNotification(
    notificationId: String, notification: Any)
case class Status()

class TradingBus(canStartAfterRegistered: Int)
    extends Actor {
    val commandHandlers =
        Map[String, Vector[CommandHandler]]()
    val notificationInterests =
        Map[String, Vector[NotificationInterest]]()

    var totalRegistered = 0

    def dispatchCommand(command: TradingCommand) = {
        if (commandHandlers.contains(command.commandId)) {
            commandHandlers(command.commandId) map {
                commandHandler =>
                    commandHandler.handler ! command.command
            }
        }
    }
}

```



```

def dispatchNotification(
  notification: TradingNotification) = {
  if (notificationInterests.contains(
    notification.notificationId)) {
    notificationInterests(
      notification.notificationId) map {
        notificationInterest =>
        notificationInterest.interested !
        notification.notification
      }
  }
}

def notifyStartWhenReady() = {
  totalRegistered += 1

  if (totalRegistered == this.canStartAfterRegistered) {
    println(s"TradingBus: is ready: $totalRegistered")
    MessageBus.canStartNow()
  }
}

def receive = {
  case register: RegisterCommandHandler =>
    println(s"TradingBus: registering: $register")
    registerCommandHandler(register.commandId,
      register.applicationId,
      register.handler)
    notifyStartWhenReady()

  case register: RegisterNotificationInterest =>
    println(s"TradingBus: registering: $register")
    registerNotificationInterest(register.notificationId,
      register.applicationId,
      register.interested)
    notifyStartWhenReady()

  case command: TradingCommand =>
    println(s"TradingBus: dispatching: $command")
    dispatchCommand(command)

  case notification: TradingNotification =>
    println(s"TradingBus: dispatching: $notification")
    dispatchNotification(notification)

  case status: Status =>
    println(s"TradingBus: STATUS: $commandHandlers")

```

```

println(s"TradingBus: STATUS: $notificationInterests")

case message: Any =>
  println(s"TradingBus: received unexpected: $message")
}

def registerCommandHandler(
  commandId: String,
  applicationId: String,
  handler: ActorRef) = {

  if (!commandHandlers.contains(commandId)) {
    commandHandlers(commandId) = Vector[CommandHandler]()
  }

  commandHandlers(commandId) =
    commandHandlers(commandId) :+
      CommandHandler(applicationId, handler)
}

def registerNotificationInterest(
  notificationId: String,
  applicationId: String,
  interested: ActorRef) = {

  if (!notificationInterests.contains(notificationId)) {
    notificationInterests(notificationId) =
      Vector[NotificationInterest]()
  }

  notificationInterests(notificationId) =
    notificationInterests(notificationId) :+
      NotificationInterest(applicationId, interested)
}

```

按照惯例，进入 TradingBus 对象的所有具体命令和通知，都必须由相应的 TradingCommand 或 TradingNotification 对象封装。这两个对象是规范化消息模型中的两个基础元素。如果某个子系统 Actor 对象尝试发送原始命令或通知，那么其发送操作就会被拒绝。然而，当命令被发送给已注册者时，就会被 TradingBus 对象解除封装并被作为原始 / 具体消息类型发送。将通知发送给已注册者时，也会执行相同的处理过程。

在本例中，只有几个具体命令和通知需要通过 TradingBus 对象发送。这些命令是 ExecuteBuyOrder 和 ExecuteSellOrder。这些通知是

BuyOrderExecuted 和 SellOrderExecuted。Actor 对象 StockTrader 将其本身注册为 ExecuteBuyOrder 和 ExecuteSellOrder 命令的处理程序，以及 BuyOrderExecuted 和 SellOrderExecuted 通知的唯一发送者。两个子系统 Actor 对象 PortfolioManager 和 MarketAnalysisTools，将它们本身注册为 BuyOrderExecuted 和 SellOrderExecuted 通知的接收者。

下面是这 3 个子系统 Actor 对象的源代码：

```
class MarketAnalysisTools(tradingBus: ActorRef) extends Actor {
  val applicationId = self.path.name

  tradingBus ! RegisterNotificationInterest(applicationId,
    "BuyOrderExecuted", self)
  tradingBus ! RegisterNotificationInterest(applicationId,
    "SellOrderExecuted", self)

  def receive = {
    case executed: BuyOrderExecuted =>
      println(s"MarketAnalysisTools: adding: $executed")
      MessageBus.completedStep()

    case executed: SellOrderExecuted =>
      println(s"MarketAnalysisTools: adjusting: $executed")
      MessageBus.completedStep()

    case message: Any =>
      println(s"MarketAnalysisTools: unexpected: $message")
  }
}
```

```
class PortfolioManager(tradingBus: ActorRef)
  extends Actor {
  val applicationId = self.path.name
```

```
  tradingBus ! RegisterNotificationInterest(
    applicationId,
    "BuyOrderExecuted", self)
  tradingBus ! RegisterNotificationInterest(
    applicationId,
    "SellOrderExecuted", self)

  def receive = {
    case executed: BuyOrderExecuted =>
      println(s"PortfolioManager: adding holding:
        $executed")
```

```

    MessageBus.completedStep()

    case executed: SellOrderExecuted =>
        println(s"PortfolioManager: adjusting holding:
            $executed")
        MessageBus.completedStep()
}

case message: Any =>
    println(s"PortfolioManager: unexpected: $message")
}

class StockTrader(tradingBus: ActorRef) extends Actor {
    val applicationId = self.path.name

    tradingBus ! RegisterCommandHandler(
        applicationId,
        "ExecuteBuyOrder",
        self)

    tradingBus ! RegisterCommandHandler(
        applicationId,
        "ExecuteSellOrder",
        self)

    def receive = {
        case buy: ExecuteBuyOrder =>
            println(s"StockTrader: buying for: $buy")
            tradingBus ! TradingNotification(
                "BuyOrderExecuted",
                BuyOrderExecuted(
                    buy.portfolioId,
                    buy.symbol,
                    buy.quantity,
                    buy.price))
            MessageBus.completedStep()

        case sell: ExecuteSellOrder =>
            println(s"StockTrader: selling for: $sell")
            tradingBus ! TradingNotification(
                "SellOrderExecuted",
                SellOrderExecuted(
                    sell.portfolioId,
                    sell.symbol,
                    sell.quantity,
                    sell.price))
            MessageBus.completedStep()
    }
}

```

```

        case message: Any =>
            println(s"StockTrader: received unexpected: $message")
        }
    }
}

```

运行这个示例应用程序时，会输出下面的结果：

```

TradingBus: dispatching:␣
  TradingCommand(ExecuteSellOrder,ExecuteSellOrder(␣
p456,MSFT,200,31.8))
StockTrader: buying for: ExecuteBuyOrder(␣
p123,MSFT,100,31.85)
TradingBus: dispatching:␣
  TradingCommand(ExecuteBuyOrder,ExecuteBuyOrder(␣
p789,MSFT,100,31.83))
StockTrader: selling for: ExecuteSellOrder(␣
p456,MSFT,200,31.8)
TradingBus: dispatching:␣
  TradingNotification(BuyOrderExecuted,BuyOrderExecuted(␣
p123,MSFT,100,31.85))
StockTrader: buying for: ExecuteBuyOrder(␣
p789,MSFT,100,31.83)
TradingBus: dispatching:␣
  TradingNotification(SellOrderExecuted,SellOrderExecuted(␣
p456,MSFT,200,31.8))
PortfolioManager: adding holding: BuyOrderExecuted(␣
p123,MSFT,100,31.85)
MarketAnalysisTools: adding: BuyOrderExecuted(␣
p123,MSFT,100,31.85)
TradingBus: dispatching:␣
  TradingNotification(BuyOrderExecuted,BuyOrderExecuted(␣
p789,MSFT,100,31.83))
PortfolioManager: adjusting holding: SellOrderExecuted(␣
p456,MSFT,200,31.8)
MarketAnalysisTools: adjusting: SellOrderExecuted(␣
p456,MSFT,200,31.8)
PortfolioManager: adding: BuyOrderExecuted(␣
p789,MSFT,100,31.83)
MarketAnalysisTools: adding: BuyOrderExecuted(␣
p789,MSFT,100,31.83)
MessageBus: is completed.

```

你现在可能已经看出，消息总线实际上就是一种基于内容的路由器。TradingBus 对象根据自己收到的消息，查找与指定消息类型对应的注册者并将消息转发给相应的注册者（TradingBus 对象还是一种服务目录或已注册服务名

单)。此外,这3个子系统 Actor 对象都是服务激活器,不论应用程序本身是否支持同步或异步访问操作,都能够使它们本身含有的应用程序服务通过消息提供给客户端。

小结

本章详细介绍了几种重要的消息通道:点对点(一对一)通道、发布—订阅(一对多)通道、数据类型(专属数据)通道、非法消息通道和死信消息通道、具有故障防护功能的确保送达机制、消息适配器(用于转换消息数据格式)和消息桥,以及担当通信支柱的消息总线。

在你掌握了 Actor 对象提供的基础消息传输功能后,本章介绍的内容为你进一步掌握更高级的响应式应用程序设计和企业级软件整合技巧,提供了更为坚实的基础。

第 6 章

消息结构

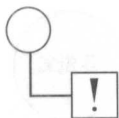
第 4 章介绍过使用 Actor 对象传递消息的方式，但是没有详细介绍应该创建和发送哪些类型的消息。任何消息都必须将发送者的意图传达给接收者。如 *Enterprise Integration Patterns* 一书中所述，可使用下列消息类型处理不同的意图。

- **消息意图**：为什么要发送消息呢？是不是要请求另一个 Actor 对象执行某个操作？如果确实如此，可使用命令消息。是不是要通知一个或多个 Actor 对象，发送者 Actor 对象已经执行了某个操作？在这种情况下，可使用事件消息。如果收到了通过消息传输大量数据的请求，可使用文档消息满足这类请求。
- **回应**：当两个 Actor 对象之间有请求—回复协定时，收到请求的 Actor 对象就需要提供回复或回应。请求由命令消息实现，而回复通常由文档消息实现。因为我们使用的是 Actor 模型，所以接收请求的 Actor 对象会知道请求发送者的地址并能够轻松做出回复。如果收到的多个请求与一个或多个彼此存在逻辑关系的回复有关，可使用相关标识符将独立的消息关联起来，形成一个逻辑消息包。
- **海量数据**：有时需要使用多个相关标识符将相关的消息关联起来。如果在关联一组消息时，还需要根据应用程序指定的次序确保这些消息的发送次序，应该怎么办呢？使用消息序列可以完成这项任务。
- **延迟的消息**：网络是不可靠的。任何类型的消息在网络中传输时，都有可能因网络延迟影响送达消息。而且，在 Actor 对象的内部也会延迟消息。例如，在处理请求消息前，Actor 对象可能还需要完成许多工作。尽管可以重新设计应用程序中的某些部分，以便使工作负载能够被及时完成，但是还是需要对这类情况设计预防措施。可使用消息有效期和死信通道，通知系统处理延迟情况。
- **消息版本**：通常，文档消息、事件消息和命令消息在它们的生命周期中会拥有多个版本。使用格式标识符可以识别消息的版本。

同理，还必须考虑在各种应用程序和整合环境中，应使用哪些种类的消息通

道，还必须设计专门用于处理响应和并发模式的消息。

命令消息



当发送消息的 Actor 对象使接收消息的 Actor 对象执行某个操作时，那么发送者 Actor 对象发送的就是命令消息。

如果你熟悉命令—查询分离原则 [CQS]，可以将命令消息视为会在被处理时对接收者产生副作用的消息，如图 6.1 所示。毕竟，命令—查询分离原则中发送命令的目的是请求转换状态。然而，*Enterprise Integration Patterns* 一书中介绍的命令消息，可能还会用于代表查询请求（即命令—查询分离原则中的查询）。因为 *Enterprise Integration Patterns* 中有意使用了重叠的概念，所以在使用命令—查询分离原则设计程序和介绍执行查询操作的消息时，最好使用更确切的术语“查询消息”。即便如此，也不是说必须使用命令—查询分离原则设计基于消息的 Actor 系统。在各种类型的系统中，使用既能够切换状态又能够获取回复的命令，可能会取得最佳效果，稍后会详细介绍这方面的内容。

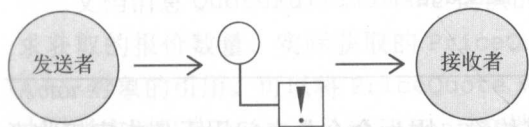


图 6.1 命令消息的发送者会命令接收者做某件事情。

任何命令消息（即使是由发送请求的 Actor 对象发送的），都是由接收者 Actor 对象的公共协定定义的。如果被发送的命令消息不符合接收者的协定，那么这条命令消息就会被转发给非法消息通道。

命令消息被设计为要执行操作的必要劝告；也就是说，劝告接收者 Actor 对象执行某个操作。命令消息可以含有任何用于执行操作的数据参数和协调 Actor 对象的参数。例如，处理传输执行命令所需的数据外，命令消息还含有返回地址，以便指明应该向哪个 Actor 对象回复执行操作产生的副作用或结果。

实际上，你可以将命令消息视为一条调用操作的语句。换言之，命令消息中含有调用操作的语句，但是允许操作在命令消息被声明后执行。

下面的样本类为交易查询领域实现了命令消息：

```
case class ExecuteBuyOrder(
  portfolioId: String,
  symbol: String,
  quantity: Int,
  price: Money)
```

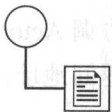
```
case class ExecuteSellOrder(
  portfolioId: String,
  symbol: String,
  quantity: Int,
  price: Money)
```

StockTrader 对象会接收两种命令消息，并且会将其他类型的消息转发给死信通道或非法消息通道：

```
class StockTrader(tradingBus: ActorRef) extends Actor {
  ...
  def receive = {
    case buy: ExecuteBuyOrder =>
      ...
    case sell: ExecuteSellOrder =>
      ...
    case message: Any =>
      context.system.deadLetters ! message
  }
}
```

通常命令消息会通过点对点通道传输，因为命令是专门用于使指定接收者 Actor 对象执行一次操作的。要发送广播类型的消息，可使用发布—订阅通道传输事件消息。

文档消息



使用文档消息可以在不指明数据处理方式的情况下，向接收者传输数据（如图 6.2 所示）。这与命令消息不同，命令消息中可能会含有数据参数，而且还会指

明数据的用途。文档消息与事件消息也不相同，事件消息不会指明数据的用途，事件消息中含有的数据代表业务领域中之前出现的某个情况。尽管文档消息也用于传输领域中的信息，但它不会指明这些信息代表业务领域中之前出现的哪些情况。要详细了解领域事件，请参阅 *Implementing Domain-Driven Design* 一书 [IDDD]。

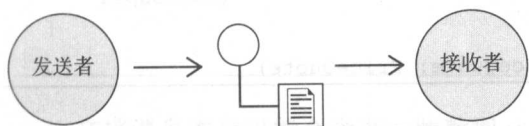


图 6.2 文档消息的发送者会为接收者提供信息，但不会指明处理这些信息的方式。

通常，文档消息可在请求—回复模式中承担回复的职责。在请求—回复模式中，文档消息接收者可能之前向文档消息的发送者发送过请求获取数据的命令消息，文档消息的发送者也可能在之前没有收到获取数据的请求的情况下，向文档消息的接收者发送文档消息。

下面的文档消息中含有满足获取报价请求的数据：

```
case class QuotationFulfillment(
  rfqId: String,
  quotesRequested: Int,
  priceQuotes: Seq[PriceQuote],
  requester: ActorRef)
```

文档消息 `QuoteFulfillment` 提供的消息包括：报价请求的唯一标识、请求获取的报价数量、实际获取的 `PriceQuote` 实例数量和请求获取报价数据的 `Actor` 对象的引用。可以将 `PriceQuote` 对象视为完整的文档消息，但是在本例中这类实例仅是 `QuoteFulfillment` 文档消息的组成部分：

```
case class PriceQuote(
  quoterId: String,
  rfqId: String,
  itemId: String,
  retailPrice: Double,
  discountPrice: Double)
```

如 `PriceQuote` 实例的结构所示，文档消息的判定条件不是消息的大小或复杂程度。更确切地说，该判定条件是不指明数据的用途（与命令消息区分开），或其中包含的数据是应用程序输出结果的组成部分（与事件消息区分开）。为了进一步了解这个差异，请观察下面的命令消息和事件消息。这些消息被组合起来，

用于获取文档消息 QuoteFulfillment:

```
case class RequestPriceQuote(
  rfqId: String,
  itemId: String,
  retailPrice: Money,
  orderTotalRetailPrice: Money)
case class PriceQuoteFulfilled(priceQuote: PriceQuote)
```

第一个样本类代表命令消息，这条消息用于请求获取指定商品的一组报价。第二个样本类代表事件消息，当某个报价满足条件时该消息就会被发布。这些消息与文档消息 QuoteFulfillment 的差异非常大。QuoteFulfillment 消息仅会含有之前被请求的报价信息。

管理处理流程和处理过程

可以使用文档消息管理处理流程和较长的处理过程。通过点对点通道在某个时刻向一个 Actor 对象发送文档消息，每条文档消息实现处理过程中的一个步骤。每个步骤完成后，都会被附到它收到的文档的后面，在该步骤完成后将变化应用到上一个处理步骤的结果中。代表当前步骤的 Actor 对象，会将已增加（或改变）内容的文档消息分派给代表下一个处理步骤的 Actor 对象。这种分派-接收-附加-分派的处理步骤会循环执行，直到整个处理过程结束。

最后一个处理步骤决定了怎样处理最终生成的文档消息。因为较长的处理过程可能是由几个或许多较短的处理过程组成的，所以最后一个处理步骤很可能仅完成较长处理过程中的一部分。在这种情况下，没有任何文档消息能够真正切换到可变状态。但是，每个处理步骤都是由当前文档和附加的信息组成的新文档消息构成的。通过执行简单的连接操作，就可以合并当前的文档消息和新数据。如果一个线性处理过程中的每个步骤都负责从指定的供货商获取 PriceQuote 实例（代表商品报价），那么使用下面的代码可以向文档消息 QuotationFulfillment 中附加数据：

```
case class QuotationFulfillment(
  rfqId: String,
  quotesRequested: Int,
  priceQuotes: Seq[PriceQuote],
  requester: ActorRef) {

  def appendWith(
    fulfilledPriceQuote: PriceQuote):
```

```

QuotationFulfillment {
  QuotationFulfillment(
    rfqId,
    quotesRequested,
    priceQuotes :+ fulfilledPriceQuote,
    requester)
}
}

```

文档消息本身可能会含有一些数据，以便描述代表每个处理步骤的 Actor 对象向下一个处理步骤分派文档消息的方式。可根据处理步骤的次序，在原始文档中添加代表每个步骤的 Actor 对象的地址和名称信息。当每个处理步骤都完成后，只需找到代表下一个处理步骤的 Actor 对象并向其分派附加了新信息的文档消息。

```

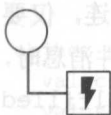
val quotationFulfillment =
  quotationFulfillment.appendWith(newPriceQuote)
quotationFulfillment.stepFollowing(name) ! quotationFulfillment

```

除了使用这种基于文档的查找方式外，还可以选择使用前面介绍过的 Akka 框架的 DistributedPubSubMediator 对象，以便能够在不查找 Actor 对象的情况下，向集群中的单个 Actor 对象分派文档消息。这种方式使用 DistributedPubSubMediator.Send 路由器消息。使用路由器消息 Send 时，只需将代表处理步骤的 Actor 对象的名称放置在文档中，无须添加这些对象的地址。DistributedPubSubMediator 对象的协定就能确保集群中符合条件的 Actor 对象，根据指定的路由策略收到需要处理的文档消息。

当较长的处理过程含有复杂的路由规范时，最好使用处理过程管理器协调向每个步骤分派的文档消息。通常，当分派规则中含有根据一个或多个步骤附加到文档消息中的数据，协调各个处理过程分支的内容时，就需要使用处理过程管理器。

事件消息



当需要通知其他 Actor 对象事件消息发送者 Actor 对象中发生的事件时，可使用图 6.3 所示的事件消息。通常，发布—订阅通道用于通知订阅方指定的事件

已经发生。然而，有时可能需要通知指定的 Actor 对象已发生的事件，或者通知指定的 Actor 对象和抽象的订阅者组会更为合适。要详细了解领域事件，请参阅 *Implementing Domain-Driven Design* 一书。

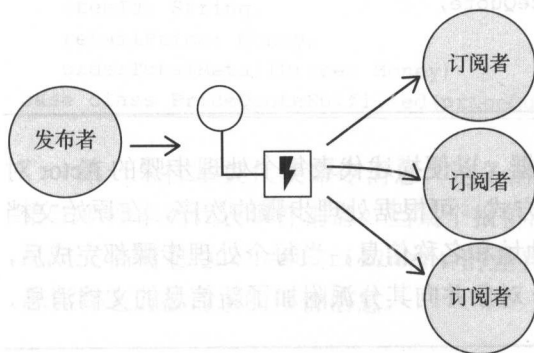


图 6.3 通过使用事件消息，发布者可以通知多个订阅者 Actor 对象领域模型中发生了哪些事件。

例如，当 OrderProcessor 对象收到请求获取报价消息 RequestForQuotation 时，它会向任意数量的商品价格折扣计算程序分派满足该请求的消息。每个选择回应该请求的商品价格折扣计算程序，都会回复含有折扣计算数据的 PriceQuote 文档消息，从而使 OrderProcessor 对象能够向聚合器发送 PriceQuoteFulfilled 事件消息。

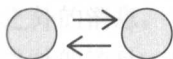
```

case class PriceQuote(
  quoterId: String,
  rfqId: String,
  itemId: String,
  retailPrice: Double,
  discountPrice: Double) // 文档消息

case class PriceQuoteFulfilled(
  priceQuote: PriceQuote) // 事件消息
  
```

在本例中，不必使用发布—订阅通道广播事件消息，因为聚合器需要了解满足报价的条件。你可以将聚合器设计为能够接收命令消息或文档消息，无法接收事件消息。而且，OrderProcessor 对象无须根据聚合器的工作方式与之相连，仅要求在 OrderProcessor 对象收到必要数量的 PriceQuoteFulfilled 事件消息时，聚合器遵守 OrderProcessor 对象的协定。还应注意，PriceQuoteFulfilled 是一种文档消息，其中含有被作为 PriceQuoteFulfilled 事件信息、通过事件消息封装的较小的 PriceQuote 文档消息。

请求—回复模式



当将消息从一个 Actor 对象发送给另一个 Actor 对象时，该消息就会被视为请求。当收到请求消息的 Actor 对象需要向请求消息的发送者发送消息时，被发送的消息就是回复。如图 6.4 所示，常见的请求—回复模式含有发送命令消息的请求者和回复文档消息的被请求者。在这类情况中，如前面介绍命令消息的内容所述，其中的命令很可能是查询消息 [IDDD]。

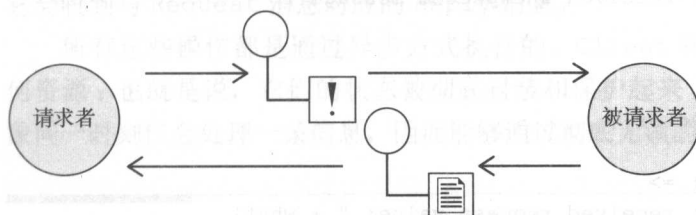


图 6.4 通过使用请求—回复模式，请求者和被请求者可以协同工作。

请求者通常会发送一条命令消息，被请求者回复文档消息并不是硬性规定。然而，如果没有比文档消息更简单的数据结构能够承担这个任务，那么使用文档消息完成回复任务通常会很合适。此处的要点是，文档消息可以运载数据，但不会规定消费者处理数据的方式。

使用 Actor 模型可以非常简单和直观地实现请求—回复模式。实际上，请求—回复模式被视为 Actor 基础语义的组成部分。下面展示了它的工作方式：

```
package co.vaughnvernon.reactiveenterprise.requestreply
```

```
import akka.actor._
```

```
import co.vaughnvernon.reactiveenterprise._
```

```
case class Request(what: String)
```

```
case class Reply(what: String)
```

```
case class StartWith(server: ActorRef)
```

```
object RequestReply extends CompletableApp(1) {
```

```
  val client = system.actorOf(Props[Client], "client")
```

```
  val server = system.actorOf(Props[Server], "server")
```

```
  client ! StartWith(server)
```

```
  awaitCompletion
```

```

println("RequestReply: is completed.")
}

class Client extends Actor {
  def receive = {
    case StartWith(server) =>
      println("Client: is starting...")
      server ! Request("REQ-1")
    case Reply(what) =>
      println("Client: received response: " + what)
      RequestReply.completedStep()
    case _ =>
      println("Client: received unexpected message")
  }
}

class Server extends Actor {
  def receive = {
    case Request(what) =>
      println("Server: received request value: " + what)
      sender ! Reply("RESP-1 for " + what)
    case _ =>
      println("Server: received unexpected message")
  }
}

```

下面是 Client 对象(代表客户端)和 Server 对象(代表服务器)输出的结果:

```

Client: is starting...
Server: received request value: REQ-1
Client: received response: RESP-1 for REQ-1
Client: is completing...

```

该程序开头的 3 个类代表可以被发送的消息。消息类型代码的后面是应用程序对象 (RequestReply)，之后是 Actor 对象 Client 和 Server。注意，在应用程序引导对象 (RequestReply) 中使用 awaitCompletion() 方法，会使该应用程序暂停运行直到这两个 Actor 对象创建完成。

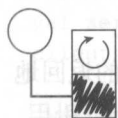
第一条消息 (StartWith) 会被发送给 Client 对象，命令该对象启动请求—回复模式。尽管 StartWith 是一种命令消息请求，但应注意 Client 对象不会生成对 RequestReply 对象的回复。StartWith 消息接收一个值为 Actor 对象 Server (实际是 ActorRef 引用) 的参数。Client 对象会向 Server 对象发送请求，而 Server 对象会向 Client 对象做出回复。Request 和 Reply 是两种消息。

Client 对象应知道怎样处理 StartWith 和 Reply 消息，而 Server 对象应知道怎样回复 Request 消息。如果 Client 对象收到了 StartWith 和 Reply 之外的消息，只需报告它无法理解这些消息。如果 Server 对象收到了 Request 之外的消息，也会使用相同的处理方式。

尽管这些细节很简单，但这个简单的 Scala/Akka 示例程序的主要目标是展示使用 Actor 模型实现请求—回复模式的方式。实现请求—回复模式非常简单。使用 Actor 模型编写程序时，请求—回复是一种自然而然被使用的模式。正如你所看到的，Server 对象不必知道它是对 Actor 对象 Client 做出的回复，它只需知道它要对 Request 消息的发送者做出回复，而 Request 消息的发送者只需知道它会收到与 Request 消息对应的 Reply 消息。

所有这些操作都是通过异步方式执行的。Client 和 Server 对象不共享任何资源；也就是说，它们的状态被彻底封装和保护起来了。而且，每个 Actor 对象同一时刻仅会处理一条信息，因而能够通过彻底无锁的异步方式处理消息。

返回地址



在使用请求—回复模式时，如果想要使请求接收者根据地址做出回复，而不是直接对请求消息的发送者做出回复，应该怎么办呢？这个问题的答案就是返回地址（如图 6.5 所示），和你选择的实现方式。

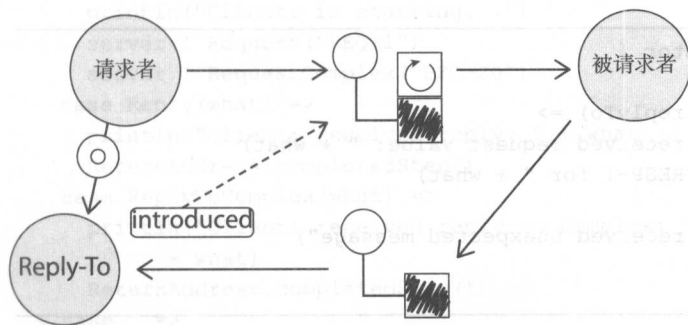


图 6.5 请求者可以使用返回地址，命令被请求者对第三方做出回复。

有趣的是，Actor 模型实际上是使用地址将消息发送给 Actor 对象的。每个 Actor 对象都有一个地址，而且要向指定的 Actor 对象发送消息，就必须知道该

Actor 对象的地址。一个 Actor 对象可以通过下列方式获得其他 Actor 对象的地址。

- 一个 Actor 对象创建了另一个 Actor 对象，因此会知道它所创建的 Actor 对象的地址。
- 一个 Actor 对象收到了消息，消息中含有其他 Actor 对象的地址。
- 有时 Actor 对象可以根据名称查找其他 Actor 对象的地址，但这样做会为 Actor 对象带来不合适的定义和实现束缚。

Enterprise Integration Patterns 中介绍的返回地址，与 Actor 模型的基础概念非常匹配。

通过指定信息提供返回地址的一种简单方式，是將你想要使之接收回复消息的 Actor 对象的地址放入发送的消息中。在前面介绍的请求—回复示例中，我们也做过类似的事情。

```
case class StartWith(server: ActorRef)
```

客户端收到的第一条消息是 StartWith，而且这条消息一定含有客户端要使用的服务器的 ActorRef 引用。这样客户端就会知道怎样向服务器发送请求。当然这样做还没有获得真正的返回地址，但你可以使用相同的方式通过消息获取返回地址。

如果客户端选择向服务器发送消息，它要提供接收回复的 Actor 对象的返回地址。当然，请求消息本身必须支持这个协议，并允许在消息中添加 ActorRef 引用。

```
case class Request(what: String, replyTo: ActorRef)
```

当服务器准备好对请求做出回复时，可以将回复消息发送给 Actor 对象 replyTo：

```
class Server extends Actor {
  def receive = {
    case Request(what, replyTo) =>
      println("Server: received request value: " + what)
      replyTo ! Reply("RESP-1 for " + what)
    case _ =>
      println("Server: received unexpected message")
  }
}
```

这种处理方式行得通，但是需要通过固定方式设计消息协议。如果出现你已经设计好了消息协议，但需要重新设计已经存在的接收者 Actor 对象，才能委托该对象的某个子 Actor 对象处理某条消息的情况，应该怎么办呢？也可能出现

某些消息含有一些复杂的处理过程，而你又不想过多增加原始 Actor 对象（如服务器）的工作负荷的情况。为服务器创建专门负责处理复杂消息的子 Actor 对象是很好的解决方案，但同时需要设计子 Actor 对象（而不是父服务器对象）对原始客户端进行回复。这样仅通过向子 Actor 对象分派工作就能够将父服务器对象解放出来，而且能够使子 Actor 对象像父服务器对象亲自完成工作一样完成分派给它的工作。

```
package co.vaughnvernon.reactiveenterprise.returnaddress
```

```
import akka.actor._
```

```
import co.vaughnvernon.reactiveenterprise._
```

```
case class Request(what: String)
```

```
case class RequestComplex(what: String)
```

```
case class Reply(what: String)
```

```
case class ReplyToComplex(what: String)
```

```
case class StartWith(server: ActorRef)
```

```
object ReturnAddress extends CompletableApp(2) {
```

```
  val client = system.actorOf(Props[Client], "client")
```

```
  val server = system.actorOf(Props[Server], "server")
```

```
  client ! StartWith(server)
```

```
  awaitCompletion
```

```
  println("ReturnAddress: is completed.")
```

```
}
```

```
class Client extends Actor {
```

```
  def receive = {
```

```
    case StartWith(server) =>
```

```
      println("Client: is starting...")
```

```
      server ! Request("REQ-1")
```

```
      server ! RequestComplex("REQ-20")
```

```
    case Reply(what) =>
```

```
      println("Client: received reply: " + what)
```

```
      ReturnAddress.completedStep()
```

```
    case ReplyToComplex(what) =>
```

```
      println("Client: received reply to complex: " + what)
```

```
      ReturnAddress.completedStep()
```

```
    case _ =>
```

```
      println("Client: received unexpected message")
```

```
  }
```

```
}
```



```

class Server extends Actor {
  val worker = context.actorOf(Props[Worker], "worker")

  def receive = {
    case request: Request =>
      println("Server: received request value: "
        + request.what)
      sender ! Reply("RESP-1 for " + request.what)
    case request: RequestComplex =>
      println("Server: received request value: "
        + request.what)
      worker forward request
    case _ =>
      println("Server: received unexpected message")
  }
}

class Worker extends Actor {
  def receive = {
    case RequestComplex(what) =>
      println("Worker: received complex request value: "
        + what)
      sender ! ReplyToComplex("RESP-2000 for " + what)
    case _ =>
      println("Worker: received unexpected message")
  }
}

```

下面是这个返回地址示例程序生成的结果：

```

Client: is starting...
Server: received request value: REQ-1
Server: received request value: REQ-20
Client: received reply: RESP-1 for REQ-1
Worker: received complex request value: REQ-20
Client: received reply to complex: RESP-2000 for REQ-20

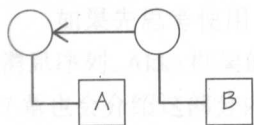
```

注意，当 Server 对象被创建时，它会使用它的上下文创建一个子 Actor 对象 Worker。只有当 Server 对象收到 RequestComplex 消息时，才会使用 Worker 对象。还应注意，不必使用第三方引用的方式设计 RequestComplex 消息。因此，只要 Client 连入网络，就会由 Server 对象处理 RequestComplex 消息。

注意，向 Worker 对象发送 RequestComplex 消息时，Server 对象不会仅告诉 Worker 对象需要执行哪些操作。更确切地说，Server 对象是将 RequestComplex 消息转发给 Worker 对象。在执行转发操作时，Worker 对象

会像直接从 Client 对象接收消息一样接收消息，这意味着 Worker 对象（而不是 Server 对象）拥有 Client 对象的地址。因此，Worker 对象像 Server 对象亲自处理工作一样处理工作。这虽然将 Server 对象解放出来，使其不再充当 Worker 对象和 Client 对象之间的中间人，但是不代表在 Worker 对象处理消息时，Server 对象能够处理其他消息。

相关标识符



通过创建相关标识符，可以使请求者和被请求者 Actor 对象将回复消息与特定的原始请求消息关联起来。具有唯一性的标识符必须既关联请求者发送的消息，又关联被请求者发送的消息，如图 6.6 所示。

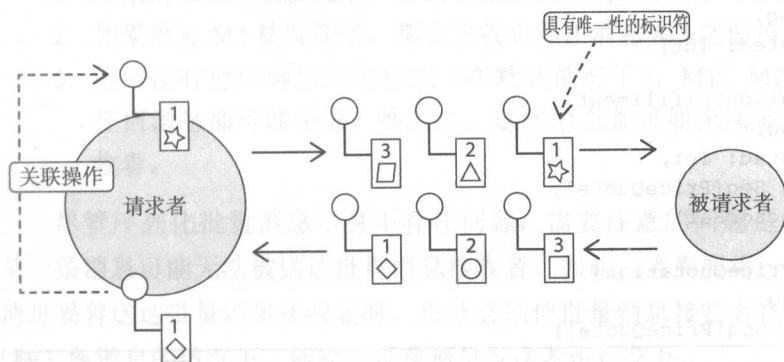


图 6.6 请求者将相关标识符附加到发出的消息中，以便使被请求者能够将回复消息与原始请求消息关联起来。

Enterprise Integration Patterns 一书在介绍相关标识符时，提出应在请求消息中创建独立的、具有唯一性的消息标识符，然后将该消息标识符用作回复消息中的相关标识符。这个具有唯一性的消息标识符通常会由消息系统生成，而且仅会被附加到消息的头部。此外，*Enterprise Integration Patterns* 一书还建议将该标识符设置为请求消息的 ID，以及回复消息中的相关 ID。

基本上，Actor 模型也是这样做的。但是，使用 Actor 模型为消息建模时，处理该标识符的方式会稍有差别。例如，除非创建组合型的消息，否则消息中没有独立的头部。因此，设计能够包含具有唯一性的业务标识符的消息类型就更为重

要。在这类情况中,无须根据不同的消息类型使用不同的名称命名标识符。实际上,使用相同的名称命名代表一类消息的标识符,通常会取得最佳效果。通过这种方式,标识符可以标识指定的业务类型。

下面的消息类型都使用 `rfqId` (代表获取报价信息请求的 ID) 进行关联:

```
case class RequestPriceQuote(
  rfqId: String,
  itemId: String,
  retailPrice: Double,
  orderTotalRetailPrice: Double)

case class PriceQuote(
  quoterId: String,
  rfqId: String,
  itemId: String,
  retailPrice: Double,
  discountPrice: Double)

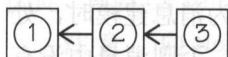
case class PriceQuoteTimedOut(rfqId: String)
case class RequiredPriceQuotesForFulfillment(
  rfqId: String,
  quotesRequested: Int)

case class QuotationFulfillment(
  rfqId: String,
  quotesRequested: Int,
  priceQuotes: Seq[PriceQuote],
  requester: ActorRef)

case class BestPriceQuotation(
  rfqId: String,
  priceQuotes: Seq[PriceQuote])
```

尽管 *Enterprise Integration Patterns* 一书着重介绍了在请求—回复模式中使用相关标识符的方式,但该标识符并非只能在这种模式中使用。例如,通过 Ad-Hoc 处理过程管理或正式的处理过程管理器,可以使用相关标识符将某项业务与较长处理过程中的所有消息关联起来。

消息序列



当需要发送必须由多条物理消息组成的一条逻辑消息时,可使用消息序列。

消息序列中的所有消息构成了一个发送批次，但是批次中的消息是作为独立元素被发送的。其中的每条消息都拥有下列成分：

- 具有唯一性的消息序列标识，如相关标识符。
- 在独立批次中指明发送次序的序列编号。序列消息的循环发送操作会执行 1 至 N 次或 0 至 $N-1$ 次， N 代表一个批次中包含的消息数量。
- 批次中的最后一条消息会含有某种标记或指示器。在一个批次中的第一条消息中放入一个计数器，也可以获得相同的效果。

如果先思考使用 Actor 模型发送和接收消息的方式，你可能会认为不必使用消息序列。Akka 框架的直接异步消息传递方式具有下列适用于消息序列的特点（第 7 章也会介绍这部分内容）：

- Actor 批量消息发送者将消息 M1、M2 和 M3 发送给批量消息接收者。

根据这个假设，会出现以下情况：

1. 如果消息 M1 被发送了，那么它就必须要在消息 M2 和 M3 之前被发送。
2. 如果消息 M2 被发送了，那么它就必须要在消息 M3 之前被发送。
3. 因为没有使用确保送达模式（在默认情况下），M1、M2 和 / 或 M3 中的任何消息都可能丢失，换言之，这些消息都可能无法被送达批量消息接收者。

尽管序列化批量消息本身不存在问题，需要注意的问题是，消息序列中的某一条消息可能无法被送达批量消息接收者。因此，当需要将一个批次中的多条消息妥善送达批量消息接收者时，你就必须使批量消息接收者在没收到批次中某（些）条消息的情况下，能够与批量消息发送者进行交互。

在设计批量消息发送者和批量消息接收者的交互操作时，最好将批量消息接收者设计为轮询消费者。在这种情况下，批量消息发送者会通过分批通信规范告诉批量消息接收者，新批次消息已经可发送。然后批量消息接收者就会根据序列消息的次序，要求获取一个批次中的每一条消息。只有当确认当前批次的消息都收到后，批量消息接收者才会要求获取下一个批次的消息。批量消息接收者可根据需要使用调度器，请求批量消息发送者重发消息，第 7 章也会介绍这方面内容。

如果批量消息发送者短时间内发送了大量的消息序列，那么批量消息接收者就必须做好准备，以便请求批量消息发送者重发它没有收到的消息序列。

消息有效期



有时可能会出现某条消息因过期而失效的情况。使用消息有效期可以控制这类超时情况,如图 6.7 所示。你可能处理过分散—聚集模式中的处理过程超时情况,但消息有效期与该情况不同。消息有效期用于确定单个消息是否已经失效,而不是用于设置完成较长的处理过程的时限。

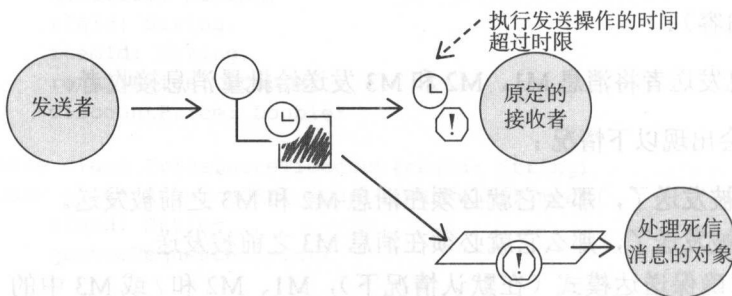


图 6.7 消息有效期被附加到会失效的消息中。

在使用基于消息的中间件时,可以在消息未被发送前,就命令消息系统使这条消息失效。当前, Akka 框架不支持能够自动检测过期消息的消息缓存。但无须担心,你自己就可以非常轻松实现这个功能。你可以创建自定义的消息缓存类型,或者直接将过期判定操作添加到消息中。这两种处理方式各有所长。下面会介绍使用特征实现这两种方式的技巧。不论消息缓存是否支持自动判定消息有效期的功能,消息本身必须支持该解决方案的部分功能。

消息的有效期应该由消息的发送者决定。毕竟,根据用户或系统的操作类型执行规范,确定消息有效期的最佳位置是消息的发送者。下面介绍具体处理方式。应先设计一个特征,使通过该特征扩展的消息设置 `timeToLive` 值。

```
trait ExpiringMessage {
  val occurredOn = System.currentTimeMillis()
  val timeToLive: Long

  def isExpired(): Boolean = {
    val elapsed = System.currentTimeMillis() - occurredOn
```

```

    elapsed > timeToLive
  }
}

```

这个特征使用创建消息的时间戳初始化了它的 `occurredOn` 字段。该特征还声明了抽象类 `timeToLive`，该类必须由扩展的具体类设置。

`ExpiringMessage` 特征还通过 `isExpired()` 方法提供了行为，以便指明消息是否过期。该操作会先获取当前的系统时间（精确到毫秒），通过减去创建完消息的时间（`occurredOn` 字段）计算出创建消息所需的时间，然后将创建消息所需的时间与客户端指定的 `timeToLive` 值做比较。

注意，这个基础算法没有考虑时区因素，你应根据自己系统的网络拓扑结构考虑该因素。至少应该考虑使各个容纳各种 Actor 对象的计算节点获得足够同步的系统时间，以便使这类计算操作能够成功执行。

下面的示例程序使用了这个特征，还定义了命令消息 `PlaceOrder`：

```

package co.vaughnvernon.reactiveenterprise.messageexpiration

```

```

import java.util.concurrent.TimeUnit
import java.util.Date
import scala.concurrent._
import scala.concurrent.duration._
import scala.util._
import ExecutionContext.Implicits.global
import akka.actor._
import co.vaughnvernon.reactiveenterprise._

```

```

case class PlaceOrder(
  id: String,
  itemId: String,
  price: Money,
  timeToLive: Long)
  extends ExpiringMessage

```

```

object MessageExpiration extends CompletableApp(3) {

```

```

  val purchaseAgent =
    system.actorOf(
      Props[PurchaseAgent],
      "purchaseAgent")

```

```

  val purchaseRouter =
    system.actorOf(
      Props(classOf[PurchaseRouter],
        purchaseAgent),
      "purchaseRouter")

```



```

purchaseRouter ! PlaceOrder("1", "11", 50.00, 1000)
purchaseRouter ! PlaceOrder("2", "22", 250.00, 100)
purchaseRouter ! PlaceOrder("3", "33", 32.95, 10)

awaitCompletion
println("MessageExpiration: is completed.")
}

```

这个示例程序创建了两个 Actor 对象：PurchaseAgent 和 PurchaseRouter。在正式的应用程序中，PurchaseRouter 对象可能会成为基于内容的路由器，根据购物消息的类型为不同类型的购物代理程序提供路由服务。本例不是要着重介绍路由功能，而是使用 PurchaseRouter 对象模拟各种原因导致的消息传输延迟。

```

class PurchaseRouter(purchaseAgent: ActorRef) extends Actor {
  val random = new Random((new Date()).getTime)

  def receive = {
    case message: Any =>
      val millis = random.nextInt(100) + 1
      println(s"PurchaseRouter: delaying delivery of"
$message for $millis milliseconds")
      val duration =
        Duration.create(millis, TimeUnit.MILLISECONDS)
      context
        .system
        .scheduler
        .scheduleOnce(duration, purchaseAgent, message)
  }
}

```

要更详细地了解 Akka 框架的 Scheduler 对象，可参阅第 7 章。

下面着重介绍使用 PurchaseAgent 对象检查消息的有效期和相应的处理方式：

```

class PurchaseAgent extends Actor {
  def receive = {
    case placeOrder: PlaceOrder =>
      if (placeOrder.isExpired()) {
        context.system.deadLetters ! placeOrder
        println(s"PurchaseAgent: delivered expired"
$placeOrder to dead letters")
      } else {
        println(s"PurchaseAgent: placing order for"
$placeOrder")
      }
  }
}

```

```
MessageExpiration.completedStep()

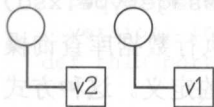
case message: Any =>
  println(s"PurchaseAgent: received unexpected:␣
$message")
}
```

如果 PlaceOrder 消息过期了, PurchaseAgent 对象就会向 Actor 系统中特殊的 Akka 框架 Actor 对象 deadLetters (该对象实现了死信通道) 发送消息。注意, *Enterprise Integration Patterns* 一书中介绍了因不同原因将过期消息发送给不同类型的消息通道的可能性, 而且这些情况的触发点都是相同的。你也可以选择彻底忽略这类消息。

下面是这个程序的运行结果:

```
PurchaseRouter: delaying delivery of PlaceOrder(␣
1,11,50.0,1000) for 87 milliseconds
PurchaseRouter: delaying delivery of PlaceOrder(␣
2,22,250.0,100) for 63 milliseconds
PurchaseRouter: delaying delivery of PlaceOrder(␣
3,33,32.95,10) for 97 milliseconds
PurchaseAgent: placing order for PlaceOrder(␣
2,22,250.0,100)
PurchaseAgent: placing order for PlaceOrder(␣
1,11,50.0,1000)
PurchaseAgent: delivered expired PlaceOrder(␣
3,33,32.95,10) to dead letters
MessageExpiration: is completed.
```

格式标识符



使用格式标识符可以为指定的消息类型设置当前的构造定义。 *Implementing Domain-Driven Design* 一书 [IDDD] 通过展示将格式标识符添加到公共语言中的方式介绍过这个技巧。

在最初定义命令消息、文档消息和事件消息时, 这些消息会含有为所有消费

者对象提供支持的所有必要信息。否则，系统将无法处理特定的消息（实际上是需要彻底实现的许多消息）。然而，仅是在较短的时间内需求也会不断改变，任何类型的消息都可能无法获取所有当前信息。这种情况不仅会在独立系统中出现，整合到一起的多个系统也会出现这类情况。

随着时间的推移，任何解决方案中消息的原始定义都不会保持不变。随着需求改变，至少部分消息必定会改变。当将新的整合系统添加到解决方案中时，必然会添加新消息，而且必然需要重新调整现存的消息。使用如图 6.8 所示的格式标识符，可以减轻系统的压力，使它们能够使用原始的或较早期的消息格式，无须更改消息格式和定义。

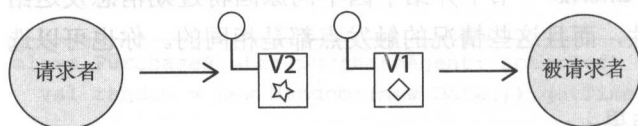


图 6.8 使用格式标识符可以为指定的消息类型设置当前的构造定义。

Enterprise Integration Patterns 一书提出，某些系统可以继续支持特定消息的原始格式。即便如此，带有更多细化的需求目标的整合参与者或子系统，会强制现存的消息类型进行升级。毕竟，参与一个开发项目的两个开发团队都很可能无法同步发布软件的日期，而使用各自的时间表工作。

格式标识符应该怎样起作用呢？*Enterprise Integration Patterns* 一书中定义了 3 种可能出现的情况，我添加了第 4 种情况。

- **版本号**：*Enterprise Integration Patterns* 一书中介绍了这种处理方式。在每条消息中嵌入整型或字符串型的版本号。该版本号使消费者系统能够根据指明的消息格式，使用不同的方式执行解除序列化或解析¹操作。通常，只要所有消息以增量（而不是减量）方式改变，消费者系统就能忽略版本号。换言之，不要从正在运行的子系统中移除当前的正确信息属性，只添加必要的新属性。
- **外键**：可以将模式文件名、文档定义或某种格式（如 `messagetype.xsd`）用作外键。外键可以是 URI/URL 或某种标识，如用于执行数据库查询操作的关键字。通过检索外键指向的内容，可以获得格式的定义。这种方式的效率可能较低，因为这要求所有消息消费者，都能够访问外键指向的位置。

¹ “解析”一词听起来好像有点不安全，*Implementing Domain-Driven Design* 一书中介绍了非常简单的、具有类型安全性的且易于维护的处理方式。

- **格式文档**：使用格式文档可以在消息中嵌入完整的格式定义（如模式）。当需要在系统之间传输含有格式文档的消息时，这种方式有明显的尺寸和传输限制缺点。
- **新扩展的消息类型**：这种方式实际上根本不会修改旧的消息格式，而会将新消息创建为旧消息格式的超集。因此，所有只使用旧消息版本的子系统会继续正常工作，而所有使用新消息的子系统，可以通过新的、专用的消息类型识别出新消息。新消息类型的名称可能会与用于扩展它的旧消息类型的名称密切关联。例如，如果旧事件消息的名称为 `OrderPlaced`，那么通过该类型消息扩展的新消息类型的名称就可以为 `OrderPlacedExt2`。在消息名称的尾部添加数字，可以指明该消息类型的改进次数。

通过扩展旧消息类型定义新消息类型的注意事项

通过扩展旧消息类型定义新消息类型（前面介绍的第4种处理方式），需要使能够识别和处理旧消息类型的子系统 Actor 对象，能够通过安全的方式忽略新类型消息。这意味着应使这些 Actor 对象仅将新类型的消息记录为警告，而不会将这些新类型的消息视为致命错误，从而避免导致中断正常的系统运行过程。要实现这种处理方式，至少在所有系统都能够支持新类型的消息前，还必须使新旧类型的消息都能够继续被发送。否则，所有只能使用旧消息类型的系统，就必须通过升级获得识别和处理最新类型消息的能力，此时再使用格式标识符就画蛇添足了。

下面的代码使用版本号处理方式改进了命令消息 `ExecuteBuyOrder`：

```
// 版本 1
case class ExecuteBuyOrder(
  portfolioId: String,
  symbol: String,
  quantity: Int,
  price: Money,
  version: Int) {
  def this(portfolioId: String, symbol: String,
    quantity: Int, price: Money)
    = this(portfolioId, symbol, quantity, price, 1)
}

// 版本 2
case class ExecuteBuyOrder(
  portfolioId: String,
```

```

symbol: String,
quantity: Int,
price: Money,
dateTimeOrdered: Date,
version: Int) {
  def this(portfolioId: String, symbol: String,
    quantity: Int, price: Money)
    = this(portfolioId, symbol, quantity,
      price, new Date(), 2)
}

```

版本 1 的 `ExecuteBuyOrder` 消息设置了 4 个业务属性：`portfolioId`、`symbol`、`quantity` 和 `price`。另一方面，版本 2 的 `ExecuteBuyOrder` 消息设置了 5 个业务属性：`portfolioId`、`symbol`、`quantity`、`price` 和 `dateTimeOrdered`。通过使用能够同时处理两个版本的 `ExecuteBuyOrder` 消息的设计，使客户端通过这两个版本的消息都能够传递 4 个参数。

```

val executeBuyOrder = ExecuteBuyOrder(portfolioId, symbol,
                                     quantity, price)

```

在版本 2 中，`dateTimeOrdered` 属性是通过重写构造器自动获得的。格式标识符 `version` 向各个消息类型中添加了额外的属性。各个版本中被重写的构造器，使你既能够使用版本标识符值 1 实例化 `ExecuteBuyOrder` 类，也能使用版本标识符值 2 实例化 `ExecuteBuyOrder` 类。

因为 `ExecuteBuyOrder` 是一种命令消息，所以可以假定定义并处理该消息的子系统，需要获得新的 `dateTimeOrdered` 属性。但是，通过为所有使用版本 1 的客户端提供合适的默认值，该系统仍然能够同时支持这两个消息版本。

```

class StockTrader(tradingBus: ActorRef) extends Actor {
  ...
  def receive = {
    case buy: ExecuteBuyOrder =>
      val orderExecutionStartedOn =
        if (buy.version == 1)
          new Date()
        else
          buy.dateTimeOrdered
    ...
  }
}

```

尽管所有使用版本 1 的客户端，都需要依靠略有误差的 `orderExecution-StartedOn` 日期时间值执行买入操作，但它们能够继续与改进的 Actor 对象 `StockTrader` 一起工作。然而，将来版本 1 的 `ExecuteBuyOrder` 消息很可能会被淘汰，所有客户端都必须更新，以便使用版本 2 的 `ExecuteBuyOrder` 消息实现近期交割。

小结

本章介绍了各种 Actor 对象能够发送和接收的消息，以及通过消息实现不同操作意图的方式。使用命令消息可以请求其他 Actor 对象执行某个操作，使用文档消息可以对查询请求做出回复，使用事件消息可以传输 Actor 系统的领域模型中发生过的情况。组合使用命令消息和文档消息，可以创建请求—回复模式。Actor 模型永远会提供 Actor 对象的返回地址，以满足请求—回复模式中回复部分的要求。本章还介绍了使用相关标识符将回复消息与指定请求关联起来的方式，以及当需要着重关注处理消息的次序时，使用消息序列的方式。当消息有使用限时时，可使用消息有效期设置消息的有效期。最后，本章介绍了使用格式标识符为消息创建多个版本的方式。

第 7 章

消息路由

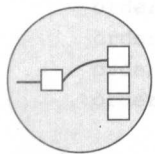
第 4 章介绍过消息路由器的基本概念，以及它在本地 Actor 系统和整合环境中的作用。路由器能够将消息资源和消息目的地分隔开。你可以通过将业务逻辑添加到任何消息路由器中，从而确定使用哪种路由方式。使用消息路由器实现业务逻辑不会有任何问题，而根据你选择的路由器类型，可以限定必要业务逻辑的数量。

因为在 Akka 集群和整合集群中都可以使用路由功能，所以应注意远程路由功能是一种处置功能。否则通过基于 Akka 的路由器只能实现消息桥，无法实现整合目标。路由器分为以下 3 类。

- **简单路由器**：包括基于内容的路由器，这种路由器能够检查消息的内容，并且能够为消息提供到达指定业务组件的路由。在仅含有 Actor 对象的系统中，基于内容的路由器仅会提供到达其他 Actor 对象的路由。消息过滤器是一种基于内容的路由器，使用这种路由器可以不为指定类型的消息提供路由，丢弃指定类型的消息。当根据应用程序的动态属性实现路由逻辑时，可使用动态路由器。使用接收者列表路由器，可以将消息发送给多个接收者，每个接收者都能够根据单个消息执行不同的操作。使用分离器可以分解单个消息，使离散函数能够根据一条消息中的不同部分执行操作。因为分离器完成了分解消息的任务，所以还需要使用聚合器将分散的结果合并到一起。因为使用分离器和聚合器时必须通过合适的次序将消息合并到一起，所以还会用到重新定序器。
- **组合路由器**：组合路由器由任意数量的简单路由器构成，并将这些简单路由器组合成一种能够完成特定工作的路由器。例如，组合消息处理器和分散一聚集器都是组合路由器，它们都能通过传送名单提供路由。
- **架构路由器**：许多路由器都能够构成粗粒度的管道和过滤器架构。消息经纪人就是一种架构路由器，这种路由器可以构成消息路由器子网中的大型网络，创建纯粹的中心辐射型架构模式。使用消息经纪人可以解决特殊类型的问题，它与消息总线类似。

Enterprise Integration Patterns 一书中介绍了一个将各种模式与适用的环境对应起来的图表。本书介绍了这些模式，你也可以参考 *Enterprise Integration Patterns* 一书中介绍的图表。

基于内容的路由器



基于内容的路由器与分离器既有相同点也有不同点。这两种路由器都用于根据消息的内容为消息提供路由，但基于内容的路由器不会像分离器那样，将一条消息分解成多条消息。更确切地说，基于内容的路由器是根据从消息内容分析出的部分逻辑，为整条消息提供路由。

此外，与消息过滤器（从消息通道中去除不需要的消息）相比，基于内容的消息也能确保不向系统传输它无法处理的消息。与此同时，基于内容的路由器还必须将系统能够处理的消息全部传输给系统。

Enterprise Integration Patterns 一书中使用了一个订单系统介绍基于内容的路由器，在该系统中基于内容的路由器必须将所有订单发送给库存系统，以便检查是否能够提供订单中的货物，如图 7.1 所示。本例假定一个订单中的所有货物都存储在某一个库存系统中。现实中可能会出现一个订单中的多种货物分别存储在多个库存系统中的情况，这就要用到分离器了。

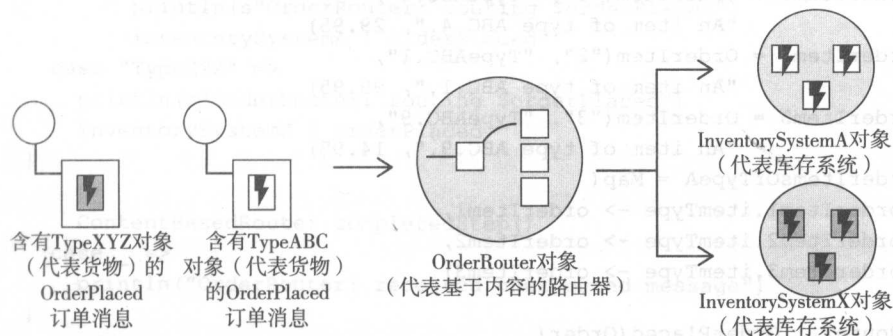


图 7.1 使用基于内容的路由器将订单消息分发给两个库存系统。

下面的例子使用基于内容的路由器检查可提供的存货：

```
package co.vaughnvernon.reactiveenterprise.contentbasedrouter
```

```
import scala.collection.Map
```

```
import akka.actor._
```

```
import co.vaughnvernon.reactiveenterprise._
```

```
case class Order(id: String, orderType: String,
                 orderItems: Map[String, OrderItem]) {
```

```
  val grandTotal: Double =
    orderItems.values.map(orderItem =>
      orderItem.price).sum
```

```
  override def toString = {
    s"Order($id, $orderType, $orderItems, $
    Totaling: $grandTotal)"
  }
}
```

```
case class OrderItem(
```

```
  id: String, itemType: String,
```

```
  description: String, price: Double) {
```

```
  override def toString = {
    s"OrderItem($id, $itemType, '$description', $price)"
  }
}
```

```
case class OrderPlaced(order: Order)
```

```
object ContentBasedRouter extends CompletableApp(3) {
```

```
  val orderRouter = system.actorOf(
    Props[OrderRouter], "orderRouter")
```

```
  val orderItem1 = OrderItem("1", "TypeABC.4",
    "An item of type ABC.4.", 29.95)
```

```
  val orderItem2 = OrderItem("2", "TypeABC.1",
    "An item of type ABC.1.", 99.95)
```

```
  val orderItem3 = OrderItem("3", "TypeABC.9",
    "An item of type ABC.9.", 14.95)
```

```
  val orderItemsOfTypeA = Map(
    orderItem1.itemType -> orderItem1,
    orderItem2.itemType -> orderItem2,
    orderItem3.itemType -> orderItem3)
```

```
  orderRouter ! OrderPlaced(Order(
    "123", "TypeABC", orderItemsOfTypeA))
```

```

val orderItem4 = OrderItem("4", "TypeXYZ.2",
    "An item of type XYZ.2.", 74.95)
val orderItem5 = OrderItem("5", "TypeXYZ.1",
    "An item of type XYZ.1.", 59.95)
val orderItem6 = OrderItem("6", "TypeXYZ.7",
    "An item of type XYZ.7.", 29.95)
val orderItem7 = OrderItem("7", "TypeXYZ.5",
    "An item of type XYZ.5.", 9.95)
val orderItemsOfTypeX = Map(
    orderItem4.itemType -> orderItem4,
    orderItem5.itemType -> orderItem5,
    orderItem6.itemType -> orderItem6,
    orderItem7.itemType -> orderItem7)

orderRouter ! OrderPlaced(Order("124", "TypeXYZ",
    orderItemsOfTypeB))

awaitCompletion
println("ContentBasedRouter: is completed.")
}

class OrderRouter extends Actor {
    val inventorySystemA =
        context.actorOf(Props[InventorySystemA],
            "inventorySystemA")
    val inventorySystemX =
        context.actorOf(Props[InventorySystemX],
            "inventorySystemX")

    def receive = {
        case orderPlaced: OrderPlaced =>
            orderPlaced.order.orderType match {
                case "TypeABC" =>
                    println(s"OrderRouter: routing $orderPlaced")
                    inventorySystemA ! orderPlaced
                case "TypeXYZ" =>
                    println(s"OrderRouter: routing $orderPlaced")
                    inventorySystemX ! orderPlaced
            }
        case _ =>
            ContentBasedRouter.completedStep()
            println("OrderRouter: received unexpected message")
    }
}

class InventorySystemA extends Actor {

```

```

def receive = {
  case OrderPlaced(order) =>
    println(s"InventorySystemA: handling $order")
    ContentBasedRouter.completedStep()
  case _ =>
    println("InventorySystemA: unexpected message")
}

class InventorySystemX extends Actor {
  def receive = {
    case OrderPlaced(order) =>
      println(s"InventorySystemX: handling $order")
      ContentBasedRouter.completedStep()
    case _ =>
      println("InventorySystemX: unexpected message")
  }
}

```

运行该程序可以获得下面的结果（此处仅列出了一部分）：

```

OrderRouter: routing OrderPlaced(Order(123, TypeABC,
Map(TypeABC.4 -> OrderItem(1, TypeABC.4, 'An item of
type ABC.4.', 29.95), TypeABC.1 -> OrderItem(2,
TypeABC.1, 'An item of type ABC.1.', 99.95), TypeABC.9
-> OrderItem(3, TypeABC.9, 'An item of type ABC.9.',
14.95)), Totaling: 144.85))
...
ContentBasedRouter: is completed.
InventorySystemX: handling Order(124, TypeXYZ, Map(
TypeXYZ.2 -> OrderItem(4, TypeXYZ.2, 'An item of type
XYZ.2.', 74.95), TypeXYZ.1 -> OrderItem(5, TypeXYZ.1,
'An item of type XYZ.1.', 59.95), TypeXYZ.7 ->
OrderItem(6, TypeXYZ.7, 'An item of type XYZ.7.',
29.95), TypeXYZ.5 -> OrderItem(7, TypeXYZ.5, 'An item
of type XYZ.5.', 9.95)), Totaling: 174.80)

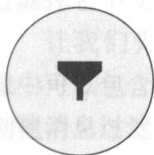
```

在本例中，基于内容的路由器检查了每条订单消息中的内容，以便决定为每条订单消息提供怎样的路由。在某些情况中，OrderRouter 对象会决定哪些货物需要在库存系统 InventorySystemA 中查找，哪些货物必须在库存系统 InventorySystemX 中查找。

在本例中，OrderRouter 对象仅通过检查 orderType 字段的值，为 OrderPlaced 消息提供到达指定库存系统的路由。但是，为什么不能使 OrderRouter

对象通过检查其他或附加的消息内容，更细致地满足路由需求呢？你可能会认为 OrderPlaced 消息本身能够执行一些帮助实现该目标的操作，使 OrderRouter 对象不必更深入地了解 OrderPlaced 消息的内容。然而，根据开发团队协作的情况，OrderPlaced 对象可能是由一支团队开发的，而 OrderRouter 对象是由另一支团队开发的，你不能要求另一支开发团队为满足路由器的要求，而在消息中专门开发特殊的功能。但是，如果 OrderPlaced 对象不是领域对象的精确副本，而是由内容过滤器或内容浓缩器生成的消息，那么该消息就能够真正地简化路由操作。

消息过滤器



当系统有可能收到它不感兴趣的或不兼容的消息，并且需要丢弃这些无用消息时，可使用消息过滤器。图 7.2 展示了这种路由模式。

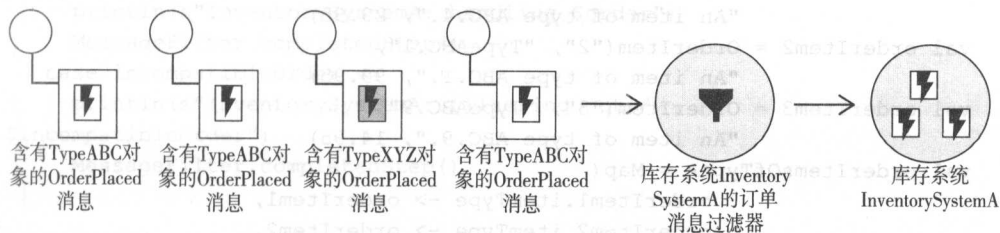


图 7.2 消息过滤器用于为库存系统丢弃无用的消息。

基于内容的路由器可以根据兼容的消息类型，为各种消息提供到达指定系统的路由。换言之，如果指定的系统支持指定类型的消息，基于内容的路由器就可以将该类型的消息传输到指定的系统。这就好像将路由器部署为消息发送系统的组成部分，或者被部署为目的地系统的起代理作用的集线器。因此，在使用基于内容的路由器时，目标系统永远都不会被发送与其处理目标不兼容的消息。

消息过滤器与基于内容的路由器不同，在使用消息过滤器的情况中，由于消息发送系统掌握的信息不完整或掌握的信息是过期的，因而不知道目标系统不能处理哪些类型的消息，所以会向目标系统发送它无法处理的消息。因此，目标系统必须先过滤掉不兼容的消息，然后才能执行核心的业务处理过程。与此同时，

目标系统中的消息过滤器必须将可用消息转发给执行核心处理操作的 Actor 对象。在这种情况下，消息过滤器好像就是目标系统，但实际上它仅是目标系统的代理。

如前面介绍基于内容的路由器的内容所述，OrderPlaced 事件消息会被发送给 InventorySystemA 和 InventorySystemX 对象。为了使用消息路由器（而不是基于内容的路由器），我们会向 InventorySystemA 和 InventorySystemX 对象发送 TypeABC 和 TypeXYZ 订单消息。下面是该示例程序的代码：

```
object MessageFilter extends CompletableApp (4) {
  val inventorySystemA =
    system.actorOf(
      Props[InventorySystemA],
      "inventorySystemA")
  val actualInventorySystemX =
    system.actorOf(
      Props[InventorySystemX],
      "inventorySystemX")
  val inventorySystemX =
    system.actorOf(
      Props(new InventorySystemXMessageFilter(
        actualInventorySystemX),
        "inventorySystemXMessageFilter")
    )
  val orderItem1 = OrderItem("1", "TypeABC.4",
    "An item of type ABC.4.", 29.95)
  val orderItem2 = OrderItem("2", "TypeABC.1",
    "An item of type ABC.1.", 99.95)
  val orderItem3 = OrderItem("3", "TypeABC.9",
    "An item of type ABC.9.", 14.95)
  val orderItemsOfTypeA = Map(
    orderItem1.itemType -> orderItem1,
    orderItem2.itemType -> orderItem2,
    orderItem3.itemType -> orderItem3)
  inventorySystemA ! OrderPlaced(Order("123", "TypeABC",
    orderItemsOfTypeA))
  inventorySystemX ! OrderPlaced(Order("123", "TypeABC",
    orderItemsOfTypeA))

  val orderItem4 = OrderItem("4", "TypeXYZ.2",
    "An item of type XYZ.2.", 74.95)
  val orderItem5 = OrderItem("5", "TypeXYZ.1",
    "An item of type XYZ.1.", 59.95)
  val orderItem6 = OrderItem("6", "TypeXYZ.7",
    "An item of type XYZ.7.", 29.95)
  val orderItem7 = OrderItem("7", "TypeXYZ.5",
    "An item of type XYZ.5.", 9.95)
  val orderItemsOfTypeX = Map(
```

```

    orderItem4.itemType -> orderItem4,
    orderItem5.itemType -> orderItem5,
    orderItem6.itemType -> orderItem6,
    orderItem7.itemType -> orderItem7)
inventorySystemA ! OrderPlaced(Order("124", "TypeXYZ",
    orderItemsOfTypeX))
inventorySystemX ! OrderPlaced(Order("124", "TypeXYZ",
    orderItemsOfTypeX))

awaitCompletion
println("MessageFilter: is completed.")
}

```

因为这两个库存系统都会接收这两种类型的消息，所以每个库存系统还必须过滤掉它不支持的消息类型。这两个库存系统会采用不同的处理方式。

让我们先看看 InventorySystemA 库存系统使用的过滤器。因为 Actor 对象中可以包含接收消息的代码块，所以可以在 Actor 对象 InventorySystemA 中创建消息过滤器：

```

class InventorySystemA extends Actor {
  def receive = {
    case OrderPlaced(order) if (order.isType("TypeABC")) =>
      println(s"InventorySystemA: handling $order")
      MessageFilter.completedStep()
    case incompatibleOrder =>
      println(s"InventorySystemA: filtering out: '$incompatibleOrder'")
      MessageFilter.completedStep()
  }
}

```

这个简单的消息过滤器会拒绝接收所有不含有 TypeABC 订单的 OrderPlaced 事件消息。然而，使用独立的 Actor 对象实现这个消息过滤器会取得更好的效果，如 InventorySystemX 库存系统中的示例：

```

class InventorySystemX extends Actor {
  def receive = {
    case OrderPlaced(order) =>
      println(s"InventorySystemX: handling $order")
      MessageFilter.completedStep()
    case _ =>
      println("InventorySystemX: unexpected message")
      MessageFilter.completedStep()
  }
}

```

```

    }
  }

class InventorySystemXMessageFilter(
  actualInventorySystemX: ActorRef)
  extends Actor {
  def receive = {
    case orderPlaced: OrderPlaced
      if (orderPlaced.order.isType("TypeXYZ")) =>
        actualInventorySystemX forward orderPlaced
        MessageFilter.completedStep()
    case incompatibleOrder =>
      println(s"InventorySystemXMessageFilter: filtering:$incompatibleOrder")
      MessageFilter.completedStep()
  }
}

```

只要这个示例程序接入了网络，InventorySystemXMessageFilter 对象就能够代表 InventorySystemX 对象，因为它能够引用 InventorySystemX 对象：

```

object MessageFilter extends CompletableApp (4) {
  ...
  val actualInventorySystemX =
    system.actorOf(
      Props[InventorySystemX],
      "inventorySystemX")
  val inventorySystemX =
    system.actorOf(
      Props(new InventorySystemXMessageFilter(
        actualInventorySystemX)),
      "inventorySystemXMessageFilter")
}

```

然而，实际上该示例程序中代表 Actor 对象 InventorySystemX 的消息过滤器，仅会转发与 InventorySystemX 系统兼容的消息并过滤掉其他消息。构造器参数 actualInventorySystemX 会被传递给该消息过滤器，该构造器参数的值为库存系统入口点 Actor 对象的引用。当收到含有 TypeXYZ 订单的 OrderPlaced 事件消息时，消息过滤器会将这类消息转发给 actualInventorySystemX 引用指向的 Actor 对象。

运行这个示例程序会得到下面的结果：

```

InventorySystemA: handling Order(123, TypeABC,
Map(TypeABC.4 -> OrderItem(1, TypeABC.4, 'An item'
of type ABC.4.', 29.95), TypeABC.1 -> OrderItem(2,

```

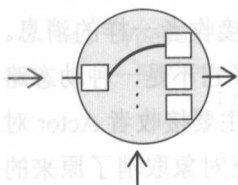
```

TypeABC.1, 'An item of type ABC.1.', 99.95)),
TypeABC.9 -> OrderItem(3, TypeABC.9, 'An item of
type ABC.9.', 14.95)), Totaling: 144.85)
InventorySystemXMessageFilter: filtering: OrderPlaced(
Order(123, TypeABC, Map(TypeABC.4 -> OrderItem(1,
TypeABC.4, 'An item of type ABC.4.', 29.95), TypeABC.1
-> OrderItem(2, TypeABC.1, 'An item of type ABC.1.',
99.95), TypeABC.9 -> OrderItem(3, TypeABC.9, 'An item
of type ABC.9.', 14.95)), Totaling: 144.85))
InventorySystemA: filtering: OrderPlaced(Order(124,
TypeXYZ, Map(TypeXYZ.2 -> OrderItem(4, TypeXYZ.2,
'An item of type XYZ.2.', 74.95), TypeXYZ.1 ->
OrderItem(5, TypeXYZ.1, 'An item of type XYZ.1.',
59.95), TypeXYZ.7 -> OrderItem(6, TypeXYZ.7, 'An
item of type XYZ.7.', 29.95), TypeXYZ.5 -> OrderItem(
7, TypeXYZ.5, 'An item of type XYZ.5.', 9.95)), Totaling:
174.79999999999998))
InventorySystemX: handling Order(124, TypeXYZ,
Map(TypeXYZ.2 -> OrderItem(4, TypeXYZ.2, 'An item
of type XYZ.2.', 74.95), TypeXYZ.1 -> OrderItem(5,
TypeXYZ.1, 'An item of type XYZ.1.', 59.95), TypeXYZ.7
-> OrderItem(6, TypeXYZ.7, 'An item of type XYZ.7.',
29.95), TypeXYZ.5 -> OrderItem(7, TypeXYZ.5, 'An item
of type XYZ.5.', 9.95)), Totaling: 174.79999999999998)
MessageFilter: is completed.

```

使用独立的 Actor 对象实现消息过滤器的一个主要优点是，可以单独维护消息过滤器而不必考虑消息过滤器代表的 Actor 对象。与此相比，每当需要引入新的消息类型或淘汰旧的消息类型时，都必须更改 Actor 对象 InventorySystemA。使用独立的 Actor 对象实现消息过滤器的缺点是，可能会略微增加系统开销，但增加的系统开销是极小的。你很可能更喜欢 InventorySystemX 示例中管道和过滤器架构提供的强大功能。

动态路由器



如果说分离器和基于内容的路由器等路由模式显得有点平凡，那么动态路由

器就是一种更具挑战性的设计。这种路由器含有多个活动部分，而且任何动态元素永远都会更为有趣。此外，动态路由器会使用规则，这些规则具有一定的复杂性。这些规则不会太复杂，但是要比分离器和基于内容的路由器使用的规则复杂。

要接收来自动态路由器的消息，Actor 对象必须注册它感兴趣的消息，如图 7.3 所示。通过最基础的注册处理过程，Actor 对象可以告诉动态路由器，它对哪种消息感兴趣。也可以通过更精细的规则，要求动态路由器执行多层次的查询操作，以便为 Actor 对象传输指定的消息。

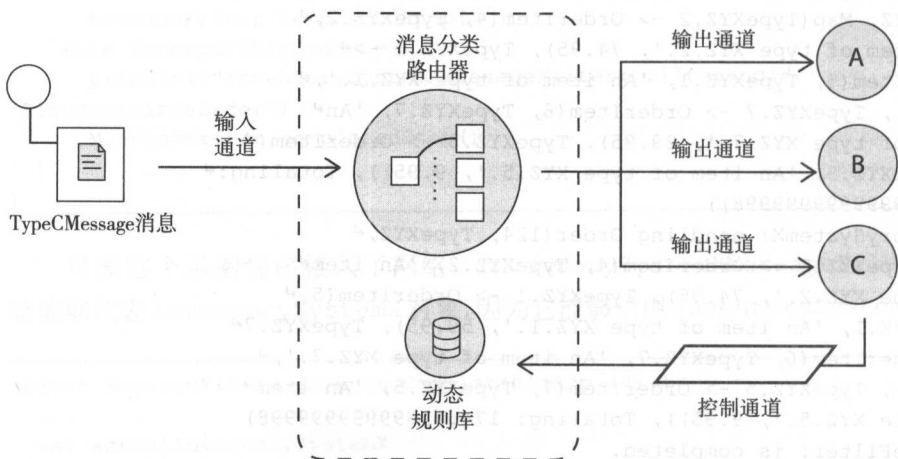


图 7.3 动态路由器仅会将指定类型的消息，传输给对该类型消息感兴趣的已注册 Actor 对象。

Enterprise Integration Patterns 一书中提出创建最新注册条目会获得最高优先权的规则。换言之，最新注册的对某种消息类型感兴趣的那个（批）Actor 对象，所有该类型的消息都会被发送给该（批）Actor 对象。在这类情况中，规则仅仅是消息的名称和一个散列表，该散列表中含有以消息类型为键，以接收者名称队列为值的键值对。这意味着大多数最新注册的接收者名称队列会替代先前注册的接收者名称队列。

下面的例子使用了相似的处理方式，仅使用了注册的 Actor 对象而没有使用队列名称。而且，如果多个 Actor 对象注册了同一种消息，动态路由器会将其他 Actor 对象视为次要接收者。次要接收者不会被发送与主要接收者一样的消息。要使多个 Actor 对象接收相同的消息，需要使用接收者列表（它不是一种动态路由器）。更确切地说，次要接收者 Actor 对象用于预防出现，主要接收者 Actor 对象不再处理原来注册的消息类型的情况。当主要接收者 Actor 对象取消了原来的注册后，次要接收者 Actor 对象就能顶替它的位置。

下面是动态路由器的控制代码，这段代码创建了多个 Actor 对象：

```
package co.vaughnvernon.reactiveenterprise.dynamicrouter

import reflect.runtime.currentMirror
import akka.actor._
import co.vaughnvernon.reactiveenterprise._

case class InterestedIn(messageType: String)
case class NoLongerInterestedIn(messageType: String)

case class TypeAMessage(description: String)
case class TypeBMessage(description: String)
case class TypeCMessage(description: String)
case class TypeDMessage(description: String)

object DynamicRouter extends CompletableApp(5) {
  val dunnoInterested =
    system.actorOf(
      Props[DunnoInterested],
      "dunnoInterested")
  val typedMessageInterestRouter =
    system.actorOf(
      Props(new TypedMessageInterestRouter(
        dunnoInterested, 4, 1)),
      "typedMessageInterestRouter")
  val typeAInterest =
    system.actorOf(
      Props(new TypeAInterested(
        typedMessageInterestRouter)),
      "typeAInterest")
  val typeBInterest =
    system.actorOf(
      Props(new TypeBInterested(
        typedMessageInterestRouter)),
      "typeBInterest")
  val typeCInterest =
    system.actorOf(
      Props(new TypeCInterested(
        typedMessageInterestRouter)),
      "typeCInterest")
  val typeCAlsoInterested =
```



```

    system.actorOf(
        Props(new TypeCAlsoInterested(
            typedMessageInterestRouter)),
        "typeCAlsoInterested")

    awaitCanStartNow()

    typedMessageInterestRouter !
        TypeAMessage("Message of TypeA.")
    typedMessageInterestRouter !
        TypeBMessage("Message of TypeB.")
    typedMessageInterestRouter !
        TypeCMessage("Message of TypeC.")

    awaitCanCompleteNow()

    typedMessageInterestRouter !
        TypeCMessage("Another message of TypeC.")
    typedMessageInterestRouter !
        TypeDMessage("Message of TypeD.")

    awaitCompletion
    println("DynamicRouter: is completed.")
}

```

注意，这段代码中有几条额外的条件等待语句，添加这些语句的原因有两个。第一个原因是，`awaitCanStartNow()` 方法能够在动态路由器 (`TypedMessageInterestRouter`) 分发消息前，为对某类消息感兴趣的 Actor 对象提供足够的注册时间。第二个原因是，`awaitCanCompleteNow()` 方法能够在 `TypeCInterested` 对象（代表对 `TypeCMessage` 类型消息感兴趣的主要接收者）取消注册，使次要接收者 `TypeCAlsoInterested` 有足够的时间替代主要接收者 `TypeCInterested`。因此，第一条 `TypeCMessage` 消息会被发送给 `TypeCInterested` 对象。第二条 `TypeCMessage` 消息会被发送给 `TypeCAlsoInterested` 对象。

这段代码不是正式的产品级代码。下面会介绍使用这段代码的原因，其主要作用是简化示例程序。

下面是运行该示例程序的结果：

```

TypeAInterested: received: TypeAMessage(Message of TypeA.)
TypeBInterested: received: TypeBMessage(Message of TypeB.)
TypeCInterested: received: TypeCMessage(Message of TypeC.)
TypeCAlsoInterested: received: TypeCMessage(Another

```

```

message of TypeC.)
DunnoInterest: received undeliverable message:
TypeDMessage(Message of TypeD.)
DynamicRouter: is completed.

```

注意，最后一条消息 `TypeDMessage` 没有被发送给专门接收这类消息的 `Actor` 对象，而是由 `Actor` 对象 `DunnoInterest` 接收了这条消息。可以将 `DunnoInterest` 视为死信（装垃圾的）`Actor` 对象。

```

class DunnoInterested extends Actor {
  def receive = {
    case message: Any =>
      println(s"DunnoInterest: received undeliverable
message: $message")
      DynamicRouter.completedStep()
  }
}

```

下面介绍对 `TypeAMessage`、`TypeBMessage` 和 `TypeCMessage` 消息感兴趣的 `Actor` 对象。在被创建后，这些 `Actor` 对象都注册了指定的消息类型，因为这些对象都获得了 `TypedMessageInterestRouter` 对象的引用 [`Actor-Endowment`]。

```

class TypeAInterested(interestRouter: ActorRef)
  extends Actor {
  interestRouter !
    InterestedIn(TypeAMessage.getClass.getName)

  def receive = {
    case message: TypeAMessage =>
      println(s"TypeAInterested: received: $message")
      DynamicRouter.completedStep()
    case message: Any =>
      println(s"TypeAInterested: unexpected: $message")
  }
}

class TypeBInterested(interestRouter: ActorRef)
  extends Actor {
  interestRouter !
    InterestedIn(TypeBMessage.getClass.getName)

  def receive = {
    case message: TypeBMessage =>
      println(s"TypeBInterested: received: $message")

```

```

        DynamicRouter.completedStep()
    case message: Any =>
        println(s"TypeBInterested: unexpected: $message")
    }
}

class TypeCInterested(interestRouter: ActorRef)
    extends Actor {
    interestRouter !
        InterestedIn(TypeCMessage.getClass.getName)

    def receive = {
        case message: TypeCMessage =>
            println(s"TypeCInterested: received: $message")

            interestRouter ! NoLongerInterestedIn(
                TypeCMessage.getClass.getName)

        DynamicRouter.completedStep()

        case message: Any =>
            println(s"TypeCInterested: unexpected: $message")
    }
}

class TypeCAlsoInterested(interestRouter: ActorRef)
    extends Actor {
    interestRouter !
        InterestedIn(TypeCMessage.getClass.getName)

    def receive = {
        case message: TypeCMessage =>
            println(s"TypeCAlsoInterested: received: $message")

            interestRouter ! NoLongerInterestedIn(
                TypeCMessage.getClass.getName)

        DynamicRouter.completedStep()

        case message: Any =>
            println(s"TypeCAlsoInterested: unexpected: $message")
    }
}

```

在这4个 Actor 对象中，有两个 Actor 对象注册了 TypeCMessage 消息。一个 Actor 对象先执行了注册操作，并成为主要接收者。另一个 Actor 对象注册为次要接收者（通常，TypeCInterested 对象会成为主要接收者，TypeCAlsoInterested

对象会成为次要接收者)。这两个 Actor 对象其中之一收到第一条 TypedMessage 消息后, 该 Actor 对象会向 TypedMessageInterestRouter 对象发送一条 NoLongerInterestedIn 消息。这会使 TypedMessageInterestRouter 对象取消该 Actor 对象的主要接收者身份, 并使用次要接收者替代它。

下面是本示例的关键之处, 动态路由器的代码:

```
import scala.collection.mutable.Map

class TypedMessageInterestRouter(
  dunnoInterested: ActorRef,
  canStartAfterRegistered: Int,
  canCompleteAfterUnregistered: Int) extends Actor {

  val interestRegistry =
    Map[String, ActorRef]()
  val secondaryInterestRegistry =
    Map[String, ActorRef]()

  def receive = {
    case interestedIn: InterestedIn =>
      registerInterest(interestedIn)
    case noLongerInterestedIn: NoLongerInterestedIn =>
      unregisterInterest(noLongerInterestedIn)
    case message: Any =>
      sendFor(message)
  }

  def registerInterest(interestedIn: InterestedIn) = {
    val messageType =
      typedMessage(interestedIn.messageType)
    if (!interestRegistry.contains(messageType)) {
      interestRegistry(messageType) = sender
    } else {
      secondaryInterestRegistry(messageType) = sender
    }

    if (interestRegistry.size +
      secondaryInterestRegistry.size
      >= canStartAfterRegistered) {
      DynamicRouter.canStartNow()
    }
  }
}
```

```

def sendFor(message: Any) = {
    val messageType =
        typeOfMessage(
            currentMirror
                .reflect(message)
                .symbol
                .toString)

    if (interestRegistry.contains(messageType)) {
        interestRegistry(messageType) forward message
    } else {
        dunnoInterested ! message
    }
}

def typeOfMessage(rawMessageType: String): String = {
    rawMessageType
        .replace('$', ' ')
        .replace('.', ' ')
        .split(' ')
        .last
        .trim
}

var unregisterCount: Int = 0

def unregisterInterest(
    noLongerInterestedIn: NoLongerInterestedIn) = {
    val messageType =
        typeOfMessage(noLongerInterestedIn.messageType)

    if (interestRegistry.contains(messageType)) {
        val wasInterested = interestRegistry(messageType)

        if (wasInterested.compareTo(sender) == 0) {
            if (secondaryInterestRegistry
                .contains(messageType)) {
                val nowInterested =
                    secondaryInterestRegistry
                        .remove(messageType)

                interestRegistry(messageType) =
                    nowInterested.get
            } else {
                interestRegistry.remove(messageType)
            }
        }
    }
}

```

```

    }
    unregisterCount = unregisterCount + 1;
    if (unregisterCount >=
        this.canCompleteAfterUnregistered) {
        DynamicRouter.canCompleteNow()
    }
}
}
}
}
}
}
}

```

这段实现代码非常直观。首先，TypedMessageInterestRouter 对象会处理 InterestedIn 消息和 NoLongerInterestedIn 消息。对指定类型消息感兴趣的 Actor 对象，可以使用这两条消息进行注册和取消注册。

最后的消息过滤器能够接收其他类型的消息，但是仅会转发已注册类型的消息。

```

...
case message: Any =>
    sendFor(message)
...
def sendFor(message: Any) = {
    val messageType =
        typeOfMessage(
            currentMirror
                .reflect(message)
                .symbol
                .toString)

    if (interestRegistry.contains(messageType)) {
        interestRegistry(messageType) forward message
    } else {
        dunnoInterested ! message
    }
}
}
}

```

所有其他类型的消息（除已注册的消息类型外），都会被发送给 DunnoInterested 对象。

本例清晰地展示了在消息排序、同步 Actor 对象和处理临时依赖关系时可能出现的困难。这个示例程序表明，通过创建同步点，可以使用 TypeCMessage 消息的次要接收者替换 TypeCMessage 消息的主要接收者。为了尽可能使示例程序简单明了，我使用 DynamicRouter.canCompleteNow() 方法完成这项任务。

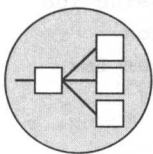
但是,在正式软件产品中不能使用这种处理方式。在这类情况中,即使主要接收者向 TypedMessageInterestRouter 对象发送了 NoLongerInterestedIn 消息,也必须能够继续接收 TypeCMessage 实例,以便提供足够的时间,使次要接收者能够被注册为主要接收者。

否则,你就必须设计新的通信协议,使 Actor 对象 TypeCInterested 和 TypeCAlsoInterested 能够切换注册状态,但这还需要使 TypedMessageInterestRouter 路由器能够管理协商操作或在其本身中包含协商逻辑。

注意

你可能会认为,因为 TypedMessageInterestRouter 对象同一时刻只能接收一条消息,所以在 TypedMessageInterestRouter 对象能够分派下一条 TypeCMessage 消息前,次要接收者(如 TypeCAlsoInterested)会替换主要接收者(TypeCInterested)。只有当在 TypedMessageInterestRouter 对象的消息缓存中, NoLongerInterestedIn 消息前面没有任何 TypeCMessage 实例时,这种假设才会成真。将这种假设(或其他类似的假设)当成现实情况,毫无疑问会出现问题。

接收者列表



接收者列表与电子邮件通讯录类似,使用通讯录可以设置任意数量的电子邮件接收者。因此,可根据将要发送的消息类型,预先确定接收者列表。但接收者列表也可以带有动态路由器的特点,通过某些业务规则确定接收者名单,如图 7.4 所示。

下面的示例程序会执行报价操作。当 MountaineeringSuppliesOrderProcessor 对象(代表登山体育用品供货商的订单处理器)收到 RequestForQuotation 消息(代表通过提供价格范围获取具体商品报价的请求)时,就会根据一系列业务规则计算出一个接收者列表,这意味着该对象是一种动态路由器。

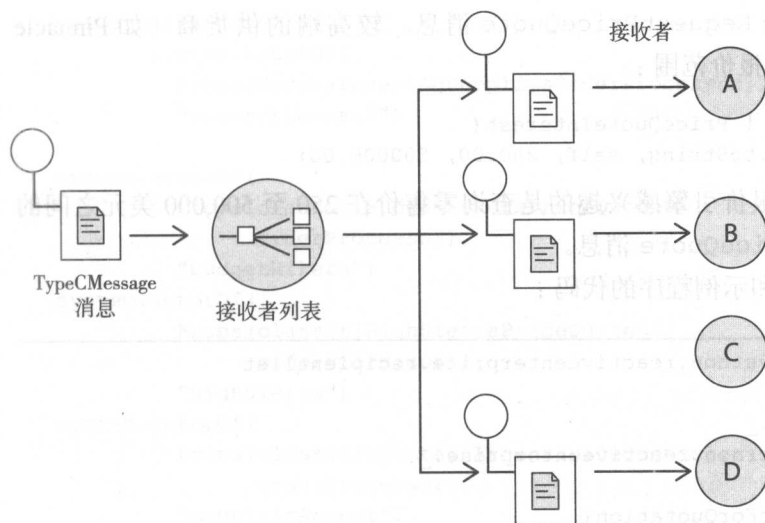


图 7.4 使用接收者列表可以确定应向哪些 Actor 对象发送指定的消息。

MountaineeringSuppliesOrderProcessor 对象会通过许多报价服务，接收 PriceQuoteInterest 消息。本示例程序中含有下列报价引擎，每个引擎都由一个 Actor 对象代表：

- BudgetHikersPriceQuotes 对象代表供货商 Budget Hikers 的报价引擎。
- HighSierraPriceQuotes 对象代表供货商 High Sierra 的报价引擎。
- MountainAscentPriceQuotes 对象代表供货商 Mountain Ascent 的报价引擎。
- PinnacleGearPriceQuotes 对象代表供货商 Pinnacle Gear 的报价引擎。
- RockBottomOuterwearPriceQuotes 对象代表供货商 Rock Bottom 的报价引擎。

这些对象被创建好后，会被赋予 MountaineeringSuppliesOrderProcessor 对象的引用。对于单个的报价引擎 Actor 对象来说，该引用就是一条注册信息。报价引擎 Actor 对象会立刻向 MountaineeringSuppliesOrderProcessor 对象发送 PriceQuoteInterest 消息（代表供货商能够提供商品的价格范围），说明它会接收哪些价格范围中的 RequestPriceQuote 消息。下面是供货商 Budget Hikers 的报价引擎发送的 PriceQuoteInterest 消息：

```
interestRegistrar ! PriceQuoteInterest(
    self.path.toString, self, 1.00, 1000.00)
```

这表明供货商 Budget Hikers 的报价引擎想要接收、查询零售价格在 1 至 1,000

美元之间的商品的 RequestPriceQuote 消息。较高端的供货商（如 Pinnacle Gear），设置了不同报价范围：

```
interestRegistrar ! PriceQuoteInterest(
    self.path.toString, self, 250.00, 500000.00)
```

Pinnacle Gear 报价引擎感兴趣的是查询零售价在 250 至 500,000 美元之间的商品的 RequestPriceQuote 消息。

下面是消息类和示例程序的代码：

```
package co.vaughnvernon.reactiveenterprise.recipientlist
```

```
import akka.actor._
```

```
import co.vaughnvernon.reactiveenterprise._
```

```
case class RequestForQuotation(
```

```
    rfqId: String,
```

```
    retailItems: Seq[RetailItem]) {
```

```
    val totalRetailPrice: Double =
```

```
        retailItems.map(retailItem =>
```

```
            retailItem.retailPrice).sum
```

```
}
```

```
case class RetailItem(
```

```
    itemId: String,
```

```
    retailPrice: Double)
```

```
case class PriceQuoteInterest(
```

```
    path: String,
```

```
    quoteProcessor: ActorRef,
```

```
    lowTotalRetail: Money,
```

```
    highTotalRetail: Money)
```

```
case class RequestPriceQuote(
```

```
    rfqId: String,
```

```
    itemId: String,
```

```
    retailPrice: Money,
```

```
    orderTotalRetailPrice: Money)
```

```
case class PriceQuote(
```

```
    rfqId: String,
```

```
    itemId: String,
```

```
    retailPrice: Money,
```

```
    discountPrice: Money)
```

```
object RecipientList extends CompletableApp(5) {
```

```

val orderProcessor =
    system.actorOf(
        Props(MountaineeringSuppliesOrderProcessor),
        "orderProcessor")

system.actorOf(
    Props(classOf[BudgetHikersPriceQuotes],
        orderProcessor),
    "budgetHikers")
system.actorOf(
    Props(classOf[HighSierraPriceQuotes],
        orderProcessor),
    "highSierra")
system.actorOf(
    Props(classOf[MountainAscentPriceQuotes],
        orderProcessor),
    "mountainAscent")
system.actorOf(
    Props(classOf[PinnacleGearPriceQuotes],
        orderProcessor),
    "pinnacleGear")
system.actorOf(
    Props(classOf[RockBottomOuterwearPriceQuotes],
        orderProcessor),
    "rockBottomOuterwear")

orderProcessor ! RequestForQuotation("123",
    Vector(RetailItem("1", 29.95),
        RetailItem("2", 99.95),
        RetailItem("3", 14.95)))

orderProcessor ! RequestForQuotation("125",
    Vector(RetailItem("4", 39.99),
        RetailItem("5", 199.95),
        RetailItem("6", 149.95),
        RetailItem("7", 724.99)))

orderProcessor ! RequestForQuotation("129",
    Vector(RetailItem("8", 119.99),
        RetailItem("9", 499.95),
        RetailItem("10", 519.00),
        RetailItem("11", 209.50)))

orderProcessor ! RequestForQuotation("135",
    Vector(RetailItem("12", 0.97),
        RetailItem("13", 9.50),
        RetailItem("14", 1.99)))

```

```

orderProcessor ! RequestForQuotation("140",
  Vector(RetailItem("15", 107.50),
    RetailItem("16", 9.50),
    RetailItem("17", 599.99),
    RetailItem("18", 249.95),
    RetailItem("19", 789.99)))
}

```

当 MountaineeringSuppliesOrderProcessor 对象收到所有 RequestForQuotation 消息时，会检查每条已注册 PriceQuoteInterest 消息的业务规则。如果某条 RequestForQuotation 消息的 totalRetailPrice 字段位于某个价格区间，那么相应的报价引擎就能够收到代表订单中商品价格的 RequestPriceQuote 消息。

```

import scala.collection.mutable.Map

class MountaineeringSuppliesOrderProcessor

  extends Actor {
    val interestRegistry = Map[String, PriceQuoteInterest]()

    def calculateRecipientList(
      rfq: RequestForQuotation): Iterable[ActorRef] = {
      for {
        interest <- interestRegistry.values
        if (rfq.totalRetailPrice >= interest.lowTotalRetail)
        if (rfq.totalRetailPrice <= interest.highTotalRetail)
      } yield interest.quoteProcessor
    }

    def dispatchTo(
      rfq: RequestForQuotation,
      recipientList: Iterable[ActorRef]) = {
      recipientList.map { recipient =>
        rfq.retailItems.map { retailItem =>
          println("OrderProcessor: "
            + rfq.rfqId
            + " item: "
            + retailItem.itemId
            + " to: "
            + recipient.path.toString)
          recipient ! RequestPriceQuote(
            rfq.rfqId,

```

```

        retailItem.itemId,
        retailItem.retailPrice,
        rfq.totalRetailPrice)
    }
  }
}

def receive = {
  case interest: PriceQuoteInterest =>
    interestRegistry(interest.path) = interest
  case priceQuote: PriceQuote =>
    println(s"OrderProcessor: received: $priceQuote")
  case rfq: RequestForQuotation =>
    val recipientList = calculateRecipientList(rfq)
    dispatchTo(rfq, recipientList)
  case message: Any =>
    println(s"OrderProcessor: unexpected: $message")
}

```

MountaineeringSuppliesOrderProcessor 对象中的 calculateRecipientList() 方法，使用 Scala 语言中的 for 推导语句，根据已经注册的业务规则确定所有接收者。获得接收者列表后，MountaineeringSuppliesOrderProcessor 对象会将 RequestForQuotation 消息中的商品报价，封装成 RequestPriceQuote 消息，发送给每个接收者。

下面是各个报价引擎的代码：

```

class BudgetHikersPriceQuotes(interestRegistrar: ActorRef)
  extends Actor {
  interestRegistrar ! PriceQuoteInterest(
    self.path.toString,
    self, 1.00, 1000.00)

  def receive = {
    case rpq: RequestPriceQuote =>
      val discount = discountPercentage(
        rpq.orderTotalRetailPrice) *
        rpq.retailPrice
      sender ! PriceQuote(rpq.rfqId, rpq.itemId,
        rpq.retailPrice,
        rpq.retailPrice - discount)
    case message: Any =>
      println(s"BudgetHikersPriceQuotes: unexpected: $message")
  }
}

```

```

def discountPercentage(
  orderTotalRetailPrice: Double) = {
  if (orderTotalRetailPrice <= 100.00) 0.02
  else if (orderTotalRetailPrice <= 399.99) 0.03
  else if (orderTotalRetailPrice <= 499.99) 0.05
  else if (orderTotalRetailPrice <= 799.99) 0.07
  else 0.075
}
}

class HighSierraPriceQuotes(interestRegistrar: ActorRef)
  extends Actor {

  interestRegistrar ! PriceQuoteInterest(
    self.path.toString, self,
    100.00, 10000.00)

  def receive = {
    case rpq: RequestPriceQuote =>
      val discount = discountPercentage(
        rpq.orderTotalRetailPrice) *
        rpq.retailPrice
      sender ! PriceQuote(rpq.rfqId, rpq.itemId,
        rpq.retailPrice,
        rpq.retailPrice - discount)

    case message: Any =>
      println(s"HighSierraPriceQuotes: unexpected: $message")
  }

  def discountPercentage(
    orderTotalRetailPrice: Double): Double = {
    if (orderTotalRetailPrice <= 150.00) 0.015
    else if (orderTotalRetailPrice <= 499.99) 0.02
    else if (orderTotalRetailPrice <= 999.99) 0.03
    else if (orderTotalRetailPrice <= 4999.99) 0.04
    else 0.05
  }
}

class MountainAscentPriceQuotes(interestRegistrar: ActorRef)
  extends Actor {
  interestRegistrar ! PriceQuoteInterest(
    self.path.toString, self,
    70.00, 5000.00)
}

```



```
def receive = {
  case rpq: RequestPriceQuote =>
    val discount = discountPercentage(
      rpq.orderTotalRetailPrice) *
      rpq.retailPrice
    sender ! PriceQuote(rpq.rfqId, rpq.itemId,
      rpq.retailPrice,
      rpq.retailPrice - discount)
  case message: Any =>
    println(s"MountainAscentPriceQuotes: unexpected:$message")
}
```

```
def discountPercentage(
  orderTotalRetailPrice: Double) = {
  if (orderTotalRetailPrice <= 99.99) 0.01
  else if (orderTotalRetailPrice <= 199.99) 0.02
  else if (orderTotalRetailPrice <= 499.99) 0.03
  else if (orderTotalRetailPrice <= 799.99) 0.04
  else if (orderTotalRetailPrice <= 999.99) 0.045
  else if (orderTotalRetailPrice <= 2999.99) 0.0475
  else 0.05
}
```

```
class PinnacleGearPriceQuotes(interestRegistrar: ActorRef)
  extends Actor {
  interestRegistrar ! PriceQuoteInterest(
    self.path.toString, self,
    250.00, 500000.00)
```

```
def receive = {
  case rpq: RequestPriceQuote =>
    val discount = discountPercentage(
      rpq.orderTotalRetailPrice) *
      rpq.retailPrice
    sender ! PriceQuote(rpq.rfqId, rpq.itemId,
      rpq.retailPrice,
      rpq.retailPrice - discount)
  case message: Any =>
    println(s"PinnacleGearPriceQuotes: unexpected:$message")
}
```

```
def discountPercentage(
  orderTotalRetailPrice: Double) = {
```

```

    if (orderTotalRetailPrice <= 299.99) 0.015
    else if (orderTotalRetailPrice <= 399.99) 0.0175
    else if (orderTotalRetailPrice <= 499.99) 0.02
    else if (orderTotalRetailPrice <= 999.99) 0.03
    else if (orderTotalRetailPrice <= 1199.99) 0.035
    else if (orderTotalRetailPrice <= 4999.99) 0.04
    else if (orderTotalRetailPrice <= 7999.99) 0.05
    else 0.06
  }
}

class RockBottomOuterwearPriceQuotes(
  interestRegistrar: ActorRef)
  extends Actor {

  interestRegistrar ! PriceQuoteInterest(
    self.path.toString, self,
    0.50, 7500.00)

  def receive = {
    case rpq: RequestPriceQuote => {
      val discount = discountPercentage(
        rpq.orderTotalRetailPrice) *
        rpq.retailPrice
      sender ! PriceQuote(rpq.rfqId, rpq.itemId,
        rpq.retailPrice,
        rpq.retailPrice - discount)
    }
    case message: Any => {
      println(s"RockBottomOuterwearPriceQuotes: $message")
      unexpected: $message"
    }
  }

  def discountPercentage(
    orderTotalRetailPrice: Double) = {
    if (orderTotalRetailPrice <= 100.00) 0.015
    else if (orderTotalRetailPrice <= 399.99) 0.02
    else if (orderTotalRetailPrice <= 499.99) 0.03
    else if (orderTotalRetailPrice <= 799.99) 0.04
    else if (orderTotalRetailPrice <= 999.99) 0.05
    else if (orderTotalRetailPrice <= 2999.99) 0.06
    else if (orderTotalRetailPrice <= 4999.99) 0.07
    else if (orderTotalRetailPrice <= 5999.99) 0.075
    else 0.08
  }
}

```

各个报价引擎之间的一个基础差异是实现 discountPercentage() 方法的

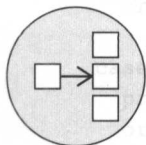
代码。我特意使这些 Actor 对象不继承抽象基类（通过抽象基类扩展）。为各种供货商提供报价服务引擎，很可能具有一些相同的特点。因此，我这样做的目的是使每个报价引擎都拥有独立的实现代码，即使它们可能很相似。

下面是执行该程序后获得的部分输出结果：

```
OrderProcessor: 123 item: 1 to: akka://mtnSupplies/
/user/rockBottomOuterwear
OrderProcessor: 123 item: 2 to: akka://mtnSupplies/
user/rockBottomOuterwear
OrderProcessor: 123 item: 3 to: akka://mtnSupplies/
user/rockBottomOuterwear
OrderProcessor: 123 item: 1 to: akka://mtnSupplies/
user/mountainAscent
...
OrderProcessor: 140 item: 19 to: akka://mtnSupplies/
user/highSierra
OrderProcessor: received: PriceQuote(123,1,29.95,29.351)
OrderProcessor: received: PriceQuote(123,2,99.95,
97.95100000000001)
OrderProcessor: received: PriceQuote(123,3,14.95,14.651)
OrderProcessor: received: PriceQuote(123,1,29.95,29.351)
OrderProcessor: received: PriceQuote(123,2,99.95,
97.95100000000001)
OrderProcessor: received: PriceQuote(123,3,14.95,14.651)
OrderProcessor: received: PriceQuote(123,1,29.95,29.0515)
OrderProcessor: received: PriceQuote(123,2,99.95,
96.95150000000001)
...
OrderProcessor: received: PriceQuote(140,19,789.99,758.3904)
```

你一定会好奇，MountaineeringSuppliesOrderProcessor 对象怎样将来自各个报价引擎的数据，重新合并为能够提供给客户的单个报价呢？要解决这个问题需要使用聚合器，接收者列表与聚合器一起构成了分散—聚集模式。

分离器



当需要将较大的消息分割成多个独立部分，并将这些独立部分作为消息发送

时，可使用分离器。分离器与基于内容的路由器有相似的地方，因为发送各个独立部分的方式是由各个独立部分的内容决定的。然而，基于内容的路由器主要用于根据消息类型将整个消息传输给指定的子系统。分离器主要用于将构成一条消息的各个独立部分，分发给不同的子系统。

下面的示例将一条 `OrderPlaced` 消息分解成多条 `Type[?]ItemOrdered` 消息：

```
package co.vaughnvernon.reactiveenterprise.splitter

import scala.collection.Map
import akka.actor._
import co.vaughnvernon.reactiveenterprise._

case class OrderItem(
  id: String,
  itemType: String,
  description: String,
  price: Money) {

  override def toString = {
    s"OrderItem($id, $itemType, '$description', $price)"
  }
}

case class Order(orderItems: Map[String, OrderItem]) {
  val grandTotal: Double =
    orderItems.values.map(_._price).sum

  override def toString = {
    s"Order(Order Items: $orderItems Totaling: $grandTotal)"
  }
}

case class OrderPlaced(order: Order)
case class TypeAItemOrdered(orderItem: OrderItem)
case class TypeBItemOrdered(orderItem: OrderItem)
case class TypeCItemOrdered(orderItem: OrderItem)

object Splitter extends CompletableApp(4) {
  val orderRouter =
    system.actorOf(
      Props[OrderRouter],
      "orderRouter")
}
```

```

val orderItem1 = OrderItem("1", "TypeA",
    "An item of type A.", 23.95)
val orderItem2 = OrderItem("2", "TypeB",
    "An item of type B.", 99.95)
val orderItem3 = OrderItem("3", "TypeC",
    "An item of type C.", 14.95)
val orderItems = Map(
    orderItem1.itemType -> orderItem1,
    orderItem2.itemType -> orderItem2,
    orderItem3.itemType -> orderItem3)

orderRouter ! OrderPlaced(Order(orderItems))

awaitCompletion
println("Splitter: is completed.")
}

class OrderRouter extends Actor {
    val orderItemTypeAProcessor = context.actorOf(
        Props[OrderItemTypeAProcessor],
        "orderItemTypeAProcessor")
    val orderItemTypeBProcessor = context.actorOf(
        Props[OrderItemTypeBProcessor],
        "orderItemTypeBProcessor")
    val orderItemTypeCProcessor = context.actorOf(
        Props[OrderItemTypeCProcessor],
        "orderItemTypeCProcessor")

    def receive = {
        case OrderPlaced(order) =>
            println(order)
            order.orderItems foreach {
                case (itemType, orderItem) => itemType match {
                    case "TypeA" =>
                        println(s"OrderRouter: routing $itemType")
                        orderItemTypeAProcessor !
                            TypeAItemOrdered(orderItem)
                    case "TypeB" =>
                        println(s"OrderRouter: routing $itemType")
                        orderItemTypeBProcessor !
                            TypeBItemOrdered(orderItem)
                    case "TypeC" =>
                        println(s"OrderRouter: routing $itemType")
                        orderItemTypeCProcessor !
                            TypeCItemOrdered(orderItem)
                }
            }
    }
}

```

```

        Splitter.completedStep()
      case _ =>
        println("OrderRouter: received unexpected message")
    }
  }

class OrderItemTypeAProcessor extends Actor {
  def receive = {
    case TypeAItemOrdered(orderItem) =>
      println(s"OrderItemTypeAProcessor: handling '$orderItem'")
      Splitter.completedStep()
    case _ =>
      println("OrderItemTypeAProcessor: unexpected")
  }
}

class OrderItemTypeBProcessor extends Actor {
  def receive = {
    case TypeBItemOrdered(orderItem) =>
      println(s"OrderItemTypeBProcessor: handling '$orderItem'")
      Splitter.completedStep()
    case _ =>
      println("OrderItemTypeBProcessor: unexpected")
  }
}

class OrderItemTypeCProcessor extends Actor {
  def receive = {
    case TypeCItemOrdered(orderItem) =>
      println(s"OrderItemTypeCProcessor: handling '$orderItem'")
      Splitter.completedStep()
    case _ =>
      println("OrderItemTypeCProcessor: unexpected")
  }
}

```

运行该程序会得到下列输出结果：

```

Order(Order Items: Map(TypeA -> OrderItem(1, 'TypeA, 'An item of type A.', 23.95), TypeB -> OrderItem(2, TypeB, 'An item of type B.', 99.95), TypeC -> OrderItem(3, TypeC, 'An item of type C.', 14.95)) Totaling: 138.85)

```

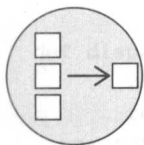
```

OrderRouter: routing TypeA
OrderRouter: routing TypeB
OrderItemTypeAProcessor: handling OrderItem(1, TypeA,
'An item of type A.', 23.95)
OrderRouter: routing TypeC
OrderItemTypeBProcessor: handling OrderItem(2, TypeB,
'An item of type B.', 99.95)
OrderItemTypeCProcessor: handling OrderItem(3, TypeC,
'An item of type C.', 14.95)
Splitter: is completed.

```

Order 对象（代表订单）含有 3 个 OrderItem 实例，每个 OrderItem 实例代表一种商品。当 OrderRouter 对象（代表分离器）收到 OrderPlaced 消息时，它会迭代每个 OrderItem 实例。根据 OrderItem 实例中 itemType 字段的值，OrderItem 实例会被封装为新的消息，并会被分发给特定类型的处理程序。

聚合器



前面介绍的接收者列表示例没有展示 MountaineeringSuppliesOrder-Processor 对象将收到的 PriceQuote 消息（代表商品报价）合并到一起的方式。要将相关的多条 PriceQuote 合并成原来的 RequestForQuotation 消息，需要使用消息中具有唯一性的 rfqId 相关标识符。

```

orderProcessor ! RequestForQuotation("123", ...)
...
recipient ! RequestPriceQuote(rfq.rfqId, ...)
...
sender ! PriceQuote(rpq.rfqId, ...)

```

本例扩展了前面介绍过的接收者列表示例，在程序中添加了聚合器，从而能够记录所有被回复的报价请求。本例将多条 PriceQuoteFulfillment 事件消息（代表完成了获取商品报价的请求—回复处理过程并获取了具体商品的报价）合并成一条 QuotationFulfillment 文档消息（代表综合商品报价），如图 7.5 所示。

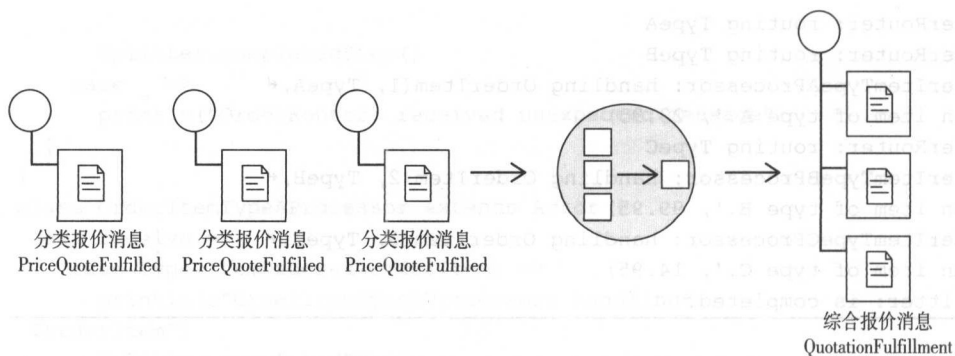


图 7.5 使用聚合器可以将多条分类报价消息合并成一条综合报价消息。

在观察下面的聚合器示例时，应先注意新的消息类型。

```
case class PriceQuoteFulfilled(priceQuote: PriceQuote)
```

```
case class RequiredPriceQuotesForFulfillment(
  rfqId: String,
  quotesRequested: Int)
```

```
case class QuotationFulfillment(
  rfqId: String,
  quotesRequested: Int,
  priceQuotes: Seq[PriceQuote],
  requester: ActorRef)
```

之后，应用程序对象创建了一个新的 Actor 对象 PriceQuoteAggregator(代表聚合器)，并将 MountaineeringSuppliesOrderProcessor 对象赋予它。

```
object Aggregator extends CompletableApp(5) {
  val priceQuoteAggregator =
    system.actorOf(
      Props[PriceQuoteAggregator],
      "priceQuoteAggregator")
```

```
  val orderProcessor = system.actorOf(
    Props(classOf[MountaineeringSuppliesOrderProcessor],
      priceQuoteAggregator),
    "orderProcessor")
  ...
```

当 MountaineeringSuppliesOrderProcessor 对象向计算出的接收

者列表中的接收者分发消息时，会通过向 PriceQuoteAggregator 对象发送 RequiredPriceQuotesForFulfillment 消息，命令 PriceQuoteAggregator 对象记录所有 PriceQuote 实例。

```
import scala.collection.mutable.Map

class MountaineeringSuppliesOrderProcessor(
  priceQuoteAggregator: ActorRef)
  extends Actor {
  val interestRegistry =
    Map[String, PriceQuoteInterest]()

  def calculateRecipientList(
    rfq: RequestForQuotation): Iterable[ActorRef] = {
    for {
      interest <- interestRegistry.values
      if (rfq.totalRetailPrice >= interest.lowTotalRetail)
      if (rfq.totalRetailPrice <= interest.highTotalRetail)
    } yield interest.quoteProcessor
  }

  def dispatchTo(
    rfq: RequestForQuotation,
    recipientList: Iterable[ActorRef]) = {
    var totalRequestedQuotes = 0
    recipientList.map { recipient =>
      rfq.retailItems.map { retailItem =>
        println("OrderProcessor: " + rfq.rfqId
          + " item: " + retailItem.itemId + " to: "
          + recipient.path.toString)
        recipient ! RequestPriceQuote(
          rfq.rfqId, retailItem.itemId,
          retailItem.retailPrice, rfq.totalRetailPrice)
      }
    }
  }

  def receive = {
    case interest: PriceQuoteInterest =>
      interestRegistry(interest.quoterId) = interest
    case priceQuote: PriceQuote =>
      priceQuoteAggregator !
        PriceQuoteFulfilled(priceQuote)
      println(s"OrderProcessor: received: $priceQuote")
    case rfq: RequestForQuotation =>
      val recipientList = calculateRecipientList(rfq)
```

```

    priceQuoteAggregator !
      RequiredPriceQuotesForFulfillment(
        rfq.rfqId,
        recipientList.size
          * rfq.retailItems.size)
    dispatchTo(rfq, recipientList)
    case fulfillment: QuotationFulfillment =>
      println(s"OrderProcessor: received: $fulfillment")
      Aggregator.completedStep()
    case message: Any =>
      println(s"OrderProcessor: unexpected: $message")
  }
}

```

PriceQuoteAggregator 对象会处理两种消息：RequiredPriceQuotesForFulfillment 和 PriceQuoteFulfilled。

```

import scala.collection.mutable.Map

class PriceQuoteAggregator extends Actor {
  val fulfilledPriceQuotes =
    Map[String, QuotationFulfillment]()

  def receive = {
    case required: RequiredPriceQuotesForFulfillment =>
      fulfilledPriceQuotes(required.rfqId) =
        QuotationFulfillment(
          required.rfqId,
          required.quotesRequested,
          Vector(),
          sender)
    case priceQuoteFulfilled: PriceQuoteFulfilled =>
      val previousFulfillment =
        fulfilledPriceQuotes(
          priceQuoteFulfilled.priceQuote.rfqId)
      val currentPriceQuotes =
        previousFulfillment.priceQuotes :+
          priceQuoteFulfilled.priceQuote
      val currentFulfillment =
        QuotationFulfillment(
          previousFulfillment.rfqId,
          previousFulfillment.quotesRequested,
          currentPriceQuotes,
          previousFulfillment.requester)

      if (currentPriceQuotes.size >=

```

```

        currentFulfillment.quotesRequested) {
    currentFulfillment.requester ! currentFulfillment
    fulfilledPriceQuotes.remove(
        priceQuoteFulfilled.priceQuote.rfqId)
} else {
    fulfilledPriceQuotes(
        priceQuoteFulfilled.priceQuote.rfqId) =
        currentFulfillment
}

println(s" PriceQuoteAggregator: fulfilled"
price quote: $priceQuoteFulfilled")
case message: Any =>
    println(s"PriceQuoteAggregator: unexpected: $message")
}
}

```

当收到 RequiredPriceQuotesForFulfillment 消息时, PriceQuoteAggregator 对象会在 fulfilledPriceQuotes 映射中创建一个新的 QuotationFulfillment 条目。收到了所有 PriceQuoteFulfilled 消息后, PriceQuoteAggregator 对象会将所有 PriceQuote 消息(被包含在 PriceQuoteFulfilled 中的), 合并成一条 QuotationFulfillment 消息。一旦 PriceQuoteAggregator 对象收到了请求获取综合报价的 PriceQuote 消息, 就会将通过合并操作生成的 QuotationFulfillment 消息发送给 OrderProcessor 对象。

```

if (currentPriceQuotes.size >=
    currentFulfillment.quotesRequested) {
    currentFulfillment.requester ! currentFulfillment
    fulfilledPriceQuotes.remove(
        priceQuoteFulfilled.priceQuote.rfqId)
} else {
    fulfilledPriceQuotes(
        priceQuoteFulfilled.priceQuote.rfqId) =
        currentFulfillment
}

```

最后, MountaineeringSuppliesOrderProcessor 对象会收到所有 QuotationFulfillment 消息。

```

class MountaineeringSuppliesOrderProcessor(
    priceQuoteAggregator: ActorRef)
    extends Actor {
...

```

```

def receive = {
  ...
  case fulfillment: QuotationFulfillment =>
    println(s"OrderProcessor: received: $fulfillment")
    Aggregator.completedStep()
  ...
}
}

```

下面是该应用程序的输出结果（此处仅列出了一部分）：

```

OrderProcessor: 123 item: 1 to: akka://default/
user/rockBottomOuterwear
OrderProcessor: 123 item: 2 to: akka://default/
user/rockBottomOuterwear
OrderProcessor: 123 item: 3 to: akka://default/
user/rockBottomOuterwear
OrderProcessor: 123 item: 1 to: akka://default/
user/mountainAscent
...
OrderProcessor: 140 item: 19 to: akka://default/
user/highSierra
OrderProcessor: received: PriceQuote(123,1,29.95,29.351)
PriceQuoteAggregator: fulfilled price quote:
  PriceQuoteFulfilled(PriceQuote(123,1,29.95,29.351))
OrderProcessor: received: PriceQuote(123,2,99.95,
97.951000000000001)
PriceQuoteAggregator: fulfilled price quote:
  PriceQuoteFulfilled(PriceQuote(123,2,99.95,
97.951000000000001))
OrderProcessor: received: PriceQuote(123,3,14.95,14.651)
OrderProcessor: received: PriceQuote(123,1,29.95,29.351)
PriceQuoteAggregator: fulfilled price quote:
  PriceQuoteFulfilled(PriceQuote(123,3,14.95,14.651))
...
PriceQuote(125,7,724.99,695.9904)),Actor[
akka://default/user/orderProcessor]
OrderProcessor: received: QuotationFulfillment(129,16,
Vector(PriceQuote(129,8,119.99,112.7906),
  PriceQuote(129,9,499.95,469.953), PriceQuote(129,10,
519.0,487.86), PriceQuote(129,11,209.5,196.93),
  PriceQuote(129,8,119.99,115.1904), PriceQuote(129,9,
499.95,479.952), PriceQuote(129,10,519.0,498.24),
  PriceQuote(129,11,209.5,201.12), PriceQuote(129,8,
119.99,114.290475), PriceQuote(129,9,499.95,
476.20237499999996), PriceQuote(129,10,519.0,494.3475),

```

```

PriceQuote(129,11,209.5,199.54875), PriceQuote(129,␣
8,119.99,115.1904), PriceQuote(129,9,499.95,479.952),␣
PriceQuote(129,10,519.0,498.24), PriceQuote(129,11,␣
209.5,201.12)), Actor[akka://default/user/orderProcessor])
OrderProcessor: received:␣
QuotationFulfillment(135,6,Vector(PriceQuote(135,12,␣
0.97,0.95545), PriceQuote(135,13,9.5,9.3575),␣
PriceQuote(135,14,1.99,1.96015), PriceQuote(135,␣
12,0.97,0.9506), PriceQuote(135,13,9.5,9.31),␣
PriceQuote(135,14,1.99,1.9502)),Actor[akka://␣
default/user/orderProcessor])
OrderProcessor: received:
QuotationFulfillment(140,20,Vector(PriceQuote(␣
140,15,107.5,101.05), PriceQuote(140,16,9.5,8.93),␣
PriceQuote(140,17,599.99,563.9906), PriceQuote(␣
140,18,249.95,234.953), PriceQuote(140,19,789.99,␣
742.5906), PriceQuote(140,15,107.5,103.2),␣
PriceQuote(140,16,9.5,9.12), PriceQuote(140,17,␣
599.99,575.9904), PriceQuote(140,18,249.95,239.952),␣
PriceQuote(140,19,789.99,758.3904), PriceQuote(140,␣
15,107.5,102.39375), PriceQuote(140,16,9.5,9.04875),␣
PriceQuote(140,17,599.99,571.4904750000001), PriceQuote␣
(140,18,249.95,238.077375), PriceQuote(140,19,789.99,␣
752.465475), PriceQuote(140,15,107.5,103.2),␣
PriceQuote(140,16,9.5,9.12), PriceQuote(140,17,␣
599.99,575.9904), PriceQuote(140,18,249.95,239.952),␣
PriceQuote(140,19,789.99,758.3904)),Actor[akka://␣
default/user/orderProcessor])
Aggregator: is completed.

```

可以使用各种结束条件设计聚合器：

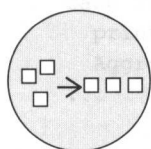
- 等待所有操作完成
- 超时
- 采用最先到达的数据
- 因重写操作导致的超时
- 外部事件

本例使用了第一个结束条件：等待所有操作完成。

实际上，将接收者列表和聚合器组合到一起就构成了分散—聚集模式。但是，使用发布—订阅通道也可以实现分散—聚集模式。

使用接收者列表和聚合器（或发布—订阅通道），可以创建较大的组合消息处理器模式。将这些路由器视为一个完整的组合消息处理器，能够更轻松地将多个组件组合成管道和过滤器模式中的独立过滤器。

重新定序器



前面介绍了许多消息路由器，这些路由器都不太复杂，但却是基础的消息传递模式。在阅读介绍接收者列表路由器的内容时，你可能会对发送消息的次序心存疑问：“怎样排定发送消息的次序呢？”

在使用 Akka 框架和其他 Actor 模型的情况中，当一个 Actor 对象直接向另一个 Actor 对象发送消息时，通常不必担心发送消息的次序。如 Akka 文档所述，一个 Actor 对象直接向另一个 Actor 对象发送的消息，永远都会以被发送的次序被接收。根据 Akka 文档，我提出下列假设：

- Actor 对象 A1 向 Actor 对象 A2 发送消息 M1、M2 和 M3。
- Actor 对象 A3 向 Actor 对象 A2 发送消息 M4、M5 和 M6。

根据上面两种可能出现的情况，可以得出下面的结论：

1. 如果要发送消息 M1，那么就必须在发送消息 M2 和 M3 前发送消息 M1。
2. 如果要发送消息 M2，那么就必须在发送消息 M3 前发送消息 M2。
3. 如果要发送消息 M4，那么就必须在发送消息 M5 和 M6 前发送消息 M4。
4. 如果要发送消息 M5，那么就必须在发送消息 M6 前发送消息 M5。
5. A2 对象会交错收到对象 A1 和 A3 发送的消息。
6. 因为没有使用确保送达机制（在默认情况下），所以任何消息都有可能丢失，即无法送达对象 A2。

此处的要点是，不必担心一个 Actor 对象直接向另一个 Actor 对象发送的消息，会以与发送次序不符的次序被接收。这种情况永远都不会出现。

然而，有时会出现由多个发送者 Actor 对象构成的复杂路由情况，一序列消息可能会以错乱的次序被接收（如前面第 5 条结论介绍的 A2 对象会交错收到对象 A1 和 A3 发送的消息）。例如，在使用基于内容的路由器（如分离器）时，就会出现这种情况。一条较大的消息会被分解成多条较小的消息，然后根据各条较小消息的内容将较小的消息发送给不同对象。处理各条较小消息所需的不同处理步骤和时间，能够轻易打乱这些消息被接收的次序。

在某些情况（如前面介绍过的分散—聚集示例）中，这个问题无关紧要。然而，

在另一些情况中，需要以发送消息的次序接收消息。要实现这个目标可使用重新定序器，如图 7.6 所示。

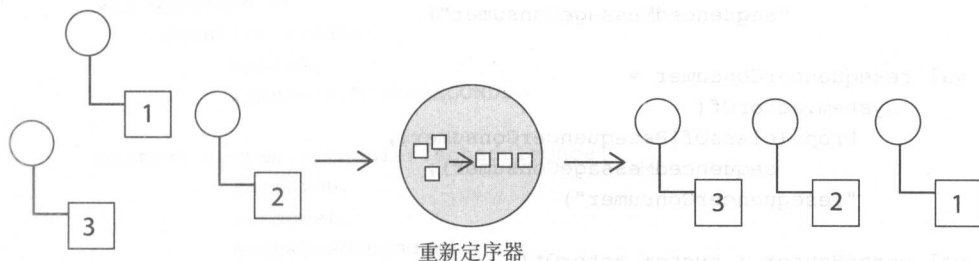


图 7.6 如果条件不允许以随机方式接收消息，可使用重新定序器为消息重新排序。

重新定序器用于接收一批打乱了原始次序的消息，然后在将这些消息发送到最终目的地前，根据必要的顺序重新排列这些消息。下面是一个使用重新定序器的示例程序：

```
package co.vaughnvernon.reactiveenterprise.resequencer

import java.util.concurrent.TimeUnit
import java.util.Date
import scala.concurrent._
import scala.concurrent.duration._
import scala.util._
import ExecutionContext.Implicits.global
import akka.actor._
import co.vaughnvernon.reactiveenterprise._

case class SequencedMessage(
  correlationId: String,
  index: Int,
  total: Int)

case class ResequencedMessages(
  dispatchableIndex: Int,
  sequencedMessages: Array[SequencedMessage]) {

  def advancedTo(dispatchableIndex: Int) = {
    ResequencedMessages(
      dispatchableIndex,
      sequencedMessages)
  }
}
```

```

object Resequencer extends CompletableApp(10) {
  val sequencedMessageConsumer = system.actorOf(
    Props[SequencedMessageConsumer],
    "sequencedMessageConsumer")

  val resequencerConsumer =
    system.actorOf(
      Props(classOf[ResequencerConsumer],
        sequencedMessageConsumer),
      "resequencerConsumer")

  val chaosRouter = system.actorOf(
    Props(classOf[ChaosRouter],
      resequencerConsumer),
    "chaosRouter")

  for (index <- 1 to 5)
    chaosRouter !
      SequencedMessage("ABC", index, 5)
  for (index <- 1 to 5)
    chaosRouter !
      SequencedMessage("XYZ", index, 5)

  awaitCompletion
  println("Resequencer: is completed.")
}

```

该程序先设置了 3 个 Actor 对象：消息的最终目的地 SequencedMessageConsumer，该对象需要以适当的次序接收消息、ResequencerConsumer 对象，该对象能够以随机次序接收消息并根据适当的顺序重新排列消息、ChaosRouter 对象，该对象以适当的次序接收消息然后以强制方式打乱它们的次序。

SequencedMessage 消息含有 correlationId 字段（该字段是指定消息顺序的索引），和 total 字段（该字段用于设置一批 SequencedMessage 消息的总数量）。correlationId 字段就是一种相关标识符。

这个示例程序先向 ChaosRouter 对象发送了多个 SequencedMessage 实例。让我们先看看这部分代码：

```

class ChaosRouter(consumer: ActorRef) extends Actor {
  val random = new Random((new Date()).getTime)

  def receive = {
    case sequencedMessage: SequencedMessage =>
      val millis = random.nextInt(100) + 1

```

```

println(s"ChaosRouter: delaying deliveryd
of $sequencedMessage for $millis milliseconds")

val duration =
  Duration.create(
    millis,
    TimeUnit.MILLISECONDS)

context.system.scheduler.scheduleOnce(
  duration,
  consumer,
  sequencedMessage)

case message: Any =>
  println(s"ChaosRouter: unexpected: $message")
}
}

```

ChaosRouter 对象被赋予了其直接消费者对象（即 ResequencerConsumer）的 ActorRef 引用。ChaosRouter 对象的基本职责是，打乱传送消息的次序。它只需设置一个 1 至 100 毫秒的随机计时器。一旦该计时器清零，ChaosRouter 对象就会将相关的 SequencedMessage 消息发送给它的消费者（即 ResequencerConsumer 对象）。让我们看看这部分代码：

```

class ResequencerConsumer(
  actualConsumer: ActorRef)
  extends Actor {
  val resequenced =
    scala.collection.mutable.Map[
      String,
      ResequencedMessages]()

  def dispatchAllSequenced(
    correlationId: String) = {
    val resequencedMessages = resequenced(correlationId)
    var dispatchableIndex =
      resequencedMessages.dispatchableIndex

    resequencedMessages.sequencedMessages.map {
      sequencedMessage =>
      if (sequencedMessage.index == dispatchableIndex) {
        actualConsumer ! sequencedMessage
      }
    }
  }
}

```

```

        dispatchableIndex += 1
    }
}

resequenced(correlationId) =
    resequencedMessages.advancedTo(dispatchableIndex)
}

def dummySequencedMessages(
    count: Int): Seq[SequencedMessage] = {
    for {
        index <- 1 to count
    } yield {
        SequencedMessage("", -1, count)
    }
}

def receive = {
    case unsequencedMessage: SequencedMessage =>
        println(s"ResequencerConsumer: received: '$unsequencedMessage'")
        resequence(unsequencedMessage)
        dispatchAllSequenced(unsequencedMessage.correlationId)
        removeCompleted(unsequencedMessage.correlationId)
    case message: Any =>
        println(s"ResequencerConsumer: unexpected: $message")
}

def removeCompleted(correlationId: String) = {
    val resequencedMessages = resequenced(correlationId)

    if (resequencedMessages.dispatchableIndex >
        resequencedMessages.sequencedMessages(0).total) {
        resequenced.remove(correlationId)
        println(s"ResequencerConsumer: removed completed: '$correlationId'")
    }
}

def resequence(
    sequencedMessage: SequencedMessage) = {
    if (!resequenced.contains(
        sequencedMessage.correlationId)) {
        resequenced(sequencedMessage.correlationId) =
            ResequencedMessages(

```

```

1,
dummySequencedMessages (
    sequencedMessage.total).toArray)
}

resequenced(sequencedMessage.correlationId)
    .sequencedMessages
    .update(sequencedMessage.index - 1,
        sequencedMessage)
}
}

```

根据 correlationId 字段的值，ResequencerConsumer 对象会使用适当的次序逐个排列它收到的每条 SequencedMessage 消息。将指定数量的 SequencedMessage 消息排好序后，ResequencerConsumer 对象就会将这些消息发送给 SequencedMessageConsumer 对象。根据这些消息被接收的次序，可能会出现一个或几个消息传输高峰。如果以 5、4、3、2、1 的次序接收消息，当 dispatchAllSequenced() 方法看到 1 号消息时，就会将 5 至 1 号消息作为一个批次，发送给 actualConsumer 引用指向的对象。

最后，让我们看看 actualConsumer 引用指向的对象 SequencedMessageConsumer(ResequencerConsumer 对象的真正消费者)。SequencedMessageConsumer 对象仅会显示它收到的每条 SequencedMessage 消息，以表明这些消息是以正确次序被接收的。

```

class SequencedMessageConsumer extends Actor {
  def receive = {
    case sequencedMessage: SequencedMessage =>
      println(s"SequencedMessageConsumer: received:␣
$sequencedMessage")
      Resequencer.completedStep()
    case message: Any =>
      println(s"SequencedMessageConsumer: unexpected:␣
$message")
  }
}

```

下面是运行这个示例程序获得的结果（此处仅列出了一部分）：

```

ChaosRouter: delaying delivery of SequencedMessage(␣
ABC,1,5) for 14 milliseconds

```

ChaosRouter: delaying delivery of SequencedMessage(ABC,2,5) for 71 milliseconds

ChaosRouter: delaying delivery of SequencedMessage(ABC,3,5) for 1 milliseconds

ChaosRouter: delaying delivery of SequencedMessage(XYZ,5,5) for 63 milliseconds

...

ResequencerConsumer: received: SequencedMessage(XYZ,4,5)

ResequencerConsumer: received: SequencedMessage(ABC,3,5)

ResequencerConsumer: received: SequencedMessage(XYZ,5,5)

ResequencerConsumer: received: SequencedMessage(ABC,5,5)

ResequencerConsumer: received: SequencedMessage(XYZ,1,5)

ResequencerConsumer: received: SequencedMessage(ABC,2,5)

SequencedMessageConsumer: received: SequencedMessage(XYZ,1,5)

ResequencerConsumer: received: SequencedMessage(XYZ,2,5)

ResequencerConsumer: received: SequencedMessage(ABC,1,5)

SequencedMessageConsumer: received: SequencedMessage(XYZ,2,5)

SequencedMessageConsumer: received: SequencedMessage(ABC,1,5)

SequencedMessageConsumer: received: SequencedMessage(ABC,2,5)

SequencedMessageConsumer: received: SequencedMessage(ABC,3,5)

ResequencerConsumer: received: SequencedMessage(XYZ,3,5)

ResequencerConsumer: removed completed: XYZ

SequencedMessageConsumer: received: SequencedMessage(XYZ,3,5)

ResequencerConsumer: received: SequencedMessage(ABC,4,5)

SequencedMessageConsumer: received: SequencedMessage(XYZ,4,5)

ResequencerConsumer: removed completed: ABC

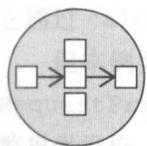
SequencedMessageConsumer: received: SequencedMessage(XYZ,5,5)

SequencedMessageConsumer: received: SequencedMessage(ABC,4,5)

SequencedMessageConsumer: received: SequencedMessage(ABC,5,5)

Resequencer: is completed.

组合消息处理器



当组合使用接收者列表和聚合器，以及基于内容的路由器和分离器时，就构成了一种较大的路由模式：组合消息处理器。将多个组件视为一个组合消息处理器，有助于将这些组件构成管道和过滤器模式中的独立过滤器。

图 7.7 展示了组合消息处理器的一些细节。

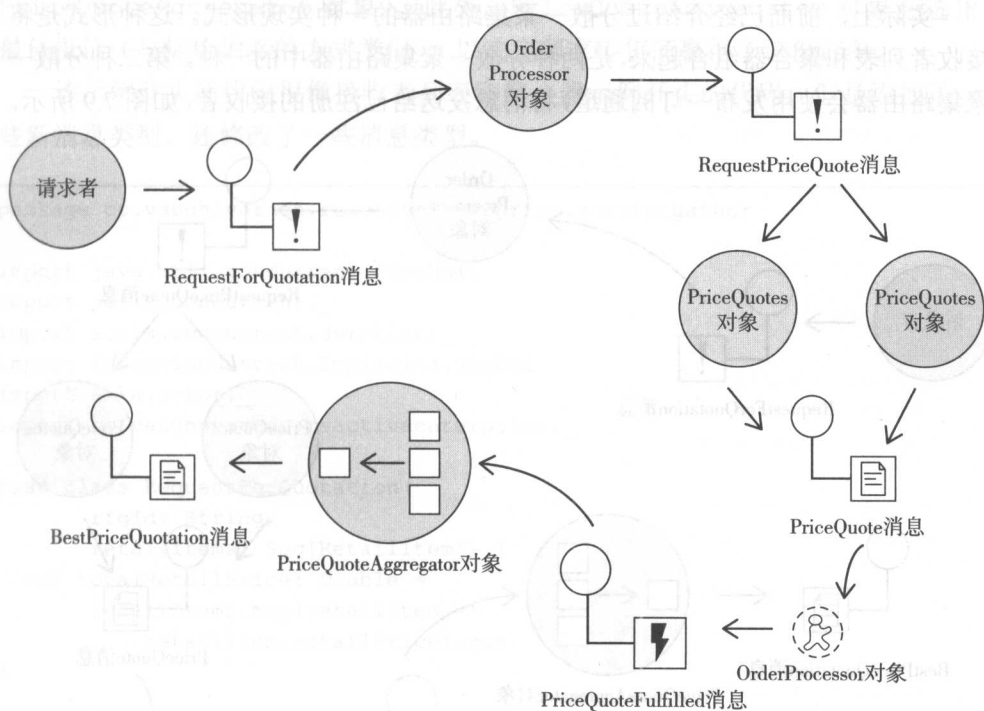


图 7.7 这个组合消息处理器是以分散—聚集模式为基础的。

将这些组件组合到一起，可以将这个组合消息处理器抽象成一个简单得多的过滤器，如图 7.8 所示。

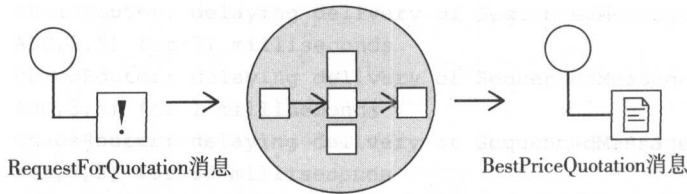


图 7.8 可以将多个分散—聚集处理组件合并成一个 Actor 对象。

要查看组合消息处理器的示例，请参阅下面介绍的分散—聚集路由器。

分散—聚集路由器

实际上，前面已经介绍过分散—聚集路由器的一种实现形式。这种形式是将接收者列表和聚合器组合起来，是两种分散—聚集路由器中的一种。第二种分散—聚集路由器会使用发布—订阅通道，将消息发送给已注册的接收者，如图 7.9 所示。

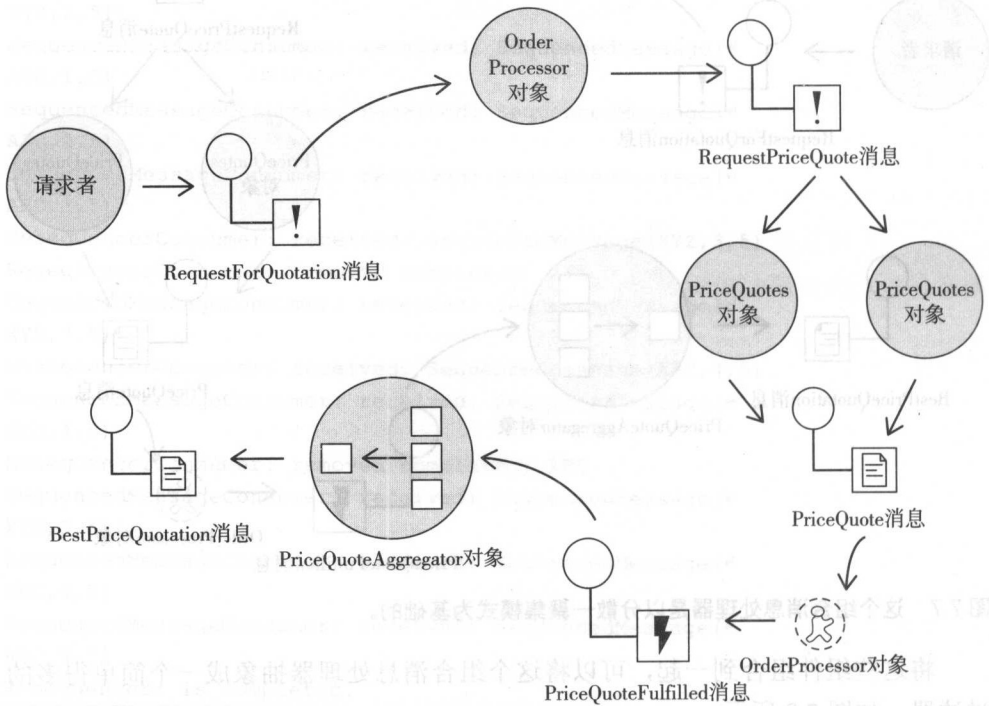


图 7.9 使用这种分散—聚集路由器可以找出最符合条件的消息。

接收者列表和聚合器示例中的 MountaineeringSuppliesOrderProcessor 对象，仍然会为对某种消息感兴趣的对象提供注册服务。当然，该对象与发布—

订阅通道不同。MountaineeringSuppliesOrderProcessor 对象不会随意为所有接收者提供报价请求消息，而是会决定哪些接收者能够获得报价请求消息。它通过 calculateRecipientList() 方法中的业务原则过滤消息。

但本例会省略使用价格区间评定各个报价引擎获取消息资格的部分。仅会将 RequestPriceQuote 消息发布给所有已注册的订阅者，并允许已注册的报价引擎提供报价。

当然，这样做会强制 MountaineeringSuppliesOrderProcessor 对象交出该处理过程中的许多控制权。但是，该对象仍然能够至少控制两个方面：时间和最佳报价。在本例中，MountaineeringSuppliesOrderProcessor 对象会使用超时设置，关闭为指定订单执行的报价操作（实际上管理超时设置的是 PriceQuoteAggregator 对象）。此外，PriceQuoteAggregator 对象会选出最佳报价（与标价拍卖的方式类似），以决定哪家零售商赢得了这份订单。

该示例开头的代码很像接收者列表和聚合器示例开头的代码，但是添加了一些新消息类型，还修改了一些消息类型。

```
package co.vaughnvernon.reactiveenterprise.scattergather
```

```
import java.util.concurrent.TimeUnit
import scala.concurrent._
import scala.concurrent.duration._
import ExecutionContext.Implicits.global
import akka.actor._
import co.vaughnvernon.reactiveenterprise._
```

```
case class RequestForQuotation(
  rfqId: String,
  retailItems: Seq[RetailItem]) {
  val totalRetailPrice: Double =
    retailItems.map(retailItem =>
      retailItem.retailPrice).sum
}
```

```
case class RetailItem(
  itemId: String,
  retailPrice: Double)
```

```
case class RequestPriceQuote(
  rfqId: String,
  itemId: String,
  retailPrice: Money,
  orderTotalRetailPrice: Money)
```

```

case class PriceQuote(
  quoteId: String,
  rfqId: String,
  itemId: String,
  retailPrice: Money,
  discountPrice: Money)

case class PriceQuoteFulfilled(priceQuote: PriceQuote)

case class PriceQuoteTimedOut(rfqId: String)

case class RequiredPriceQuotesForFulfillment(
  rfqId: String,
  quotesRequested: Int)

case class QuotationFulfillment(
  rfqId: String,
  quotesRequested: Int,
  priceQuotes: Seq[PriceQuote],
  requester: ActorRef)

case class BestPriceQuotation(
  rfqId: String,
  priceQuotes: Seq[PriceQuote])

case class SubscribeToPriceQuoteRequests(
  quoteId: String,
  quoteProcessor: ActorRef)

object ScatterGather extends CompletableApp(5) {
  val priceQuoteAggregator =
    system.actorOf(
      Props[PriceQuoteAggregator],
      "priceQuoteAggregator")

  val orderProcessor =
    system.actorOf(
      Props(
        classOf[MountaineeringSuppliesOrderProcessor],
        priceQuoteAggregator,
        "orderProcessor")

    system.actorOf(
      Props(BudgetHikersPriceQuotes,
        orderProcessor),
      "budgetHikers")

```

```

system.actorOf(
    Props(classOf[HighSierraPriceQuotes],
        orderProcessor),
    "highSierra")

system.actorOf(
    Props(classOf[MountainAscentPriceQuotes],
        orderProcessor),
    "mountainAscent")

system.actorOf(
    Props(classOf[PinnacleGearPriceQuotes],
        orderProcessor),
    "pinnacleGear")

system.actorOf(
    Props(classOf[RockBottomOuterwearPriceQuotes],
        orderProcessor),
    "rockBottomOuterwear")

orderProcessor ! RequestForQuotation("123",
    Vector(RetailItem("1", 29.95),
        RetailItem("2", 99.95),
        RetailItem("3", 14.95)))

orderProcessor ! RequestForQuotation("125",
    Vector(RetailItem("4", 39.99),
        RetailItem("5", 199.95),
        RetailItem("6", 149.95),
        RetailItem("7", 724.99)))

orderProcessor ! RequestForQuotation("129",
    Vector(RetailItem("8", 119.99),
        RetailItem("9", 499.95),
        RetailItem("10", 519.00),
        RetailItem("11", 209.50)))

orderProcessor ! RequestForQuotation("135",
    Vector(RetailItem("12", 0.97),
        RetailItem("13", 9.50),
        RetailItem("14", 1.99)))

orderProcessor ! RequestForQuotation("140",
    Vector(RetailItem("15", 1295.50),
        RetailItem("16", 9.50),
        RetailItem("17", 599.99),

```

```

        RetailItem("18", 249.95),
        RetailItem("19", 789.99)))

    awaitCompletion
    println("Scatter-Gather: is completed.")
}

```

请注意消息类型 `BestPriceQuotation`，这类消息是最终发送给 `MountaineeringSuppliesOrderProcessor` 对象的消息。`MountaineeringSuppliesOrderProcessor` 对象收到 5 条这种消息（每条消息都与 `MountaineeringSuppliesOrderProcessor` 对象收到的 `RequestForQuotation` 消息对应）后，本示例程序就会结束运行。

本例 `MountaineeringSuppliesOrderProcessor` 对象中的 `dispatch()` 方法，会将 `RequestPriceQuote` 消息发送给所有订阅者，而不是仅将 `RequestPriceQuote` 消息发送给符合某种业务规则的报价处理器。

```

import scala.collection.mutable.Map

class MountaineeringSuppliesOrderProcessor(
    priceQuoteAggregator: ActorRef)
    extends Actor {
    val subscribers =
        Map[String, SubscribeToPriceQuoteRequests]()

    def dispatch(rfq: RequestForQuotation) = {
        subscribers.values.map { subscriber =>
            val quoteProcessor = subscriber.quoteProcessor
            rfq.retailItems.map { retailItem =>
                println("OrderProcessor:"
                    + rfq.rfqId
                    + " item: "
                    + retailItem.itemId
                    + " to: "
                    + subscriber.quoterId)
                quoteProcessor !
                    RequestPriceQuote(
                        rfq.rfqId,
                        retailItem.itemId,
                        retailItem.retailPrice,
                        rfq.totalRetailPrice)
            }
        }
    }
}

```

```

def receive = {
  case subscriber: SubscribeToPriceQuoteRequests =>
    subscribers(subscriber.path) = subscriber
  case priceQuote: PriceQuote =>
    priceQuoteAggregator !
      PriceQuoteFulfilled(priceQuote)
    println(s"OrderProcessor: received: $priceQuote")
  case rfq: RequestForQuotation =>
    priceQuoteAggregator !
      RequiredPriceQuotesForFulfillment(
        rfq.rfqId,
        subscribers.size * rfq.retailItems.size)
    dispatch(rfq)
  case bestPriceQuotation: BestPriceQuotation =>
    println(s"OrderProcessor: received:$bestPriceQuotation")
    ScatterGather.completedStep()
  case message: Any =>
    println(s"OrderProcessor: unexpected: $message")
}
}

```

BestPriceQuotation 消息含有通过多个报价引擎提供的 PriceQuote 实例合并出的最佳报价。下面是该聚合器的代码。

```

import scala.collection.mutable.Map

class PriceQuoteAggregator extends Actor {
  val fulfilledPriceQuotes =
    Map[String, QuotationFulfillment]()

  def bestPriceQuotationFrom(
    quotationFulfillment: QuotationFulfillment)
    : BestPriceQuotation = {
    val bestPrices = Map[String, PriceQuote]()

    quotationFulfillment.priceQuotes.map { priceQuote =>
      if (bestPrices.contains(priceQuote.itemId)) {
        if (bestPrices(priceQuote.itemId).discountPrice >
            priceQuote.discountPrice) {
          bestPrices(priceQuote.itemId) = priceQuote
        }
      } else {
        bestPrices(priceQuote.itemId) = priceQuote
      }
    }
  }
}

```

```

    }

    BestPriceQuotation(
        quotationFulfillment.rfqId,
        bestPrices.values.toVector()
    )
}

def receive = {
    case required: RequiredPriceQuotesForFulfillment =>
        fulfilledPriceQuotes(required.rfqId) =
            QuotationFulfillment(
                required.rfqId,
                required.quotesRequested,
                Vector(),
                sender)

    val duration = Duration.create(2, TimeUnit.SECONDS)

    context.system.scheduler.scheduleOnce(
        duration, self,
        PriceQuoteTimedOut(required.rfqId))

    case priceQuoteFulfilled: PriceQuoteFulfilled =>
        priceQuoteRequestFulfilled(priceQuoteFulfilled)
        println(s"PriceQuoteAggregator: fulfilled price
quote: $PriceQuoteFulfilled")

    case priceQuoteTimedOut: PriceQuoteTimedOut =>
        priceQuoteRequestTimedOut(priceQuoteTimedOut.rfqId)

    case message: Any =>
        println(s"PriceQuoteAggregator: unexpected: $message")
}

def priceQuoteRequestFulfilled(
    priceQuoteFulfilled: PriceQuoteFulfilled) = {
    if (fulfilledPriceQuotes.contains(
        priceQuoteFulfilled.priceQuote.rfqId)) {
        val previousFulfillment =
            fulfilledPriceQuotes(
                priceQuoteFulfilled.priceQuote.rfqId)
        val currentPriceQuotes =
            previousFulfillment.priceQuotes :+
            priceQuoteFulfilled.priceQuote
        val currentFulfillment =
            QuotationFulfillment(
                previousFulfillment.rfqId,
                previousFulfillment.quotesRequested,

```



```

        currentPriceQuotes,
        previousFulfillment.requester)

    if (currentPriceQuotes.size >=
        currentFulfillment.quotesRequested) {
        quoteBestPrice(currentFulfillment)
    } else {
        fulfilledPriceQuotes(
            priceQuoteFulfilled.priceQuote.rfqId) =
            currentFulfillment
    }
}

def priceQuoteRequestTimedOut(rfqId: String) = {
    if (fulfilledPriceQuotes.contains(rfqId)) {
        quoteBestPrice(fulfilledPriceQuotes(rfqId))
    }
}

def quoteBestPrice(
    quotationFulfillment: QuotationFulfillment) = {
    if (fulfilledPriceQuotes.contains(
        quotationFulfillment.rfqId)) {
        quotationFulfillment.requester !
            bestPriceQuotationFrom(quotationFulfillment)
        fulfilledPriceQuotes.remove(
            quotationFulfillment.rfqId)
    }
}
}

```

PriceQuoteAggregator 对象管理了多个重要的操作。注意，当回应 RequiredPriceQuotesForFulfillment 消息时，该对象使用了超时设置，以便限定处理每条 RequestForQuotation 消息所用的时间。本例使用的时限为 2 秒，你可以根据具体的业务更改该设置。当出现超时情况时，PriceQuoteAggregator 对象会收到 PriceQuoteTimedOut 消息，相应 RequestForQuotation 消息处理过程就会结束。终止该处理过程不代表出了故障。更确切地说，这表明 PriceQuoteAggregator 对象会根据它收到的 PriceQuoteFulfilled 消息的数量，为 MountaineeringSuppliesOrderProcessor 对象提供一条 BestPriceQuotation 消息。

下面让我们看看 quoteBestPrice() 和 bestPriceQuotationFrom() 操作。在合并操作完成后，BestPriceQuotation 消息是由 quoteBestPrice()

方法发送给 MountaineeringSuppliesOrderProcessor 对象的。而且，quoteBestPrice() 方法使用 bestPriceQuotationFrom() 方法，通过仅提取含有最低报价的 PriceQuote 实例，创建了 BestPriceQuotation 消息。

最后，让我们看看报价引擎。这些报价引擎大多与前面介绍过的报价引擎相同，本例仅略微修改了其中的两个。BudgetHikersPriceQuotes 和 RockBottomOuterwearPriceQuotes 对象都会分别忽略报价高于 1,000 美元和 2,000 美元的订单。这会使 MountaineeringSuppliesOrderProcessor 对象无法收到预定数量的 PriceQuote 消息（和使 PriceQuoteAggregator 对象无法收到相应数量的 PriceQuoteFulfilled 消息），进而导致某些报价的处理过程超时。

```
class BudgetHikersPriceQuotes(
  priceQuoteRequestPublisher: ActorRef)
  extends Actor {
  val quoterId = self.path.name

  priceQuoteRequestPublisher !
    SubscribeToPriceQuoteRequests(quoterId, self)

  def receive = {
    case rpq: RequestPriceQuote =>
      if (rpq.orderTotalRetailPrice < 1000.00) {
        val discount = discountPercentage(
          rpq.orderTotalRetailPrice) *
          rpq.retailPrice
        sender ! PriceQuote(
          quoterId,
          rpq.rfqId,
          rpq.itemId,
          rpq.retailPrice,
          rpq.retailPrice - discount)
      } else {
        println(s"BudgetHikersPriceQuotes: ignoring: $rpq")
      }

    case message: Any =>
      println(s"BudgetHikersPriceQuotes: unexpected: $message")
  }

  def discountPercentage(
    orderTotalRetailPrice: Double) = {
    if (orderTotalRetailPrice <= 100.00) 0.02
```

```

else if (orderTotalRetailPrice <= 399.99) 0.03
else if (orderTotalRetailPrice <= 499.99) 0.05
else if (orderTotalRetailPrice <= 799.99) 0.07
else 0.075
}
}

class HighSierraPriceQuotes(
  priceQuoteRequestPublisher: ActorRef)
  extends Actor {
    val quoterId = self.path.name

    priceQuoteRequestPublisher !
      SubscribeToPriceQuoteRequests(quoterId, self)

    def receive = {
      case rpq: RequestPriceQuote =>
        val discount = discountPercentage(
          rpq.orderTotalRetailPrice) *
          rpq.retailPrice

        sender ! PriceQuote(
          quoterId,
          rpq.rfqId,
          rpq.itemId,
          rpq.retailPrice,
          rpq.retailPrice - discount)

      case message: Any =>
        println(s"HighSierraPriceQuotes: unexpected: '$message'")
    }

    def discountPercentage(
      orderTotalRetailPrice: Double): Double = {
      if (orderTotalRetailPrice <= 150.00) 0.015
      else if (orderTotalRetailPrice <= 499.99) 0.02
      else if (orderTotalRetailPrice <= 999.99) 0.03
      else if (orderTotalRetailPrice <= 4999.99) 0.04
      else 0.05
    }
  }

class MountainAscentPriceQuotes(
  priceQuoteRequestPublisher: ActorRef)
  extends Actor {
    val quoterId = self.path.name

```

```

priceQuoteRequestPublisher !
    SubscribeToPriceQuoteRequests(quoterId, self)

def receive = {
  case rpq: RequestPriceQuote =>
    val discount = discountPercentage(
      rpq.orderTotalRetailPrice) *
      rpq.retailPrice
    sender ! PriceQuote(
      quoterId,
      rpq.rfqId,
      rpq.itemId,
      rpq.retailPrice,
      rpq.retailPrice - discount)

  case message: Any =>
    println(s"MountainAscentPriceQuotes: unexpected:$message")
}

def discountPercentage(
  orderTotalRetailPrice: Double) = {
  if (orderTotalRetailPrice <= 99.99) 0.01
  else if (orderTotalRetailPrice <= 199.99) 0.02
  else if (orderTotalRetailPrice <= 499.99) 0.03
  else if (orderTotalRetailPrice <= 799.99) 0.04
  else if (orderTotalRetailPrice <= 999.99) 0.045
  else if (orderTotalRetailPrice <= 2999.99) 0.0475
  else 0.05
}

class PinnacleGearPriceQuotes(
  priceQuoteRequestPublisher: ActorRef)
  extends Actor {
  val quoterId = self.path.name

  priceQuoteRequestPublisher !
    SubscribeToPriceQuoteRequests(quoterId, self)

  def receive = {
    case rpq: RequestPriceQuote =>
      val discount = discountPercentage(
        rpq.orderTotalRetailPrice) *
        rpq.retailPrice
      sender ! PriceQuote(

```

```

        quoterId,
        rpq.rfqId,
        rpq.itemId,
        rpq.retailPrice,
        rpq.retailPrice - discount)

    case message: Any =>
        println(s"PinnacleGearPriceQuotes: unexpected: $message")
    }

    def discountPercentage(
        orderTotalRetailPrice: Double) = {
        if (orderTotalRetailPrice <= 299.99) 0.015
        else if (orderTotalRetailPrice <= 399.99) 0.0175
        else if (orderTotalRetailPrice <= 499.99) 0.02
        else if (orderTotalRetailPrice <= 999.99) 0.03
        else if (orderTotalRetailPrice <= 1199.99) 0.035
        else if (orderTotalRetailPrice <= 4999.99) 0.04
        else if (orderTotalRetailPrice <= 7999.99) 0.05
        else 0.06
    }
}

class RockBottomOuterwearPriceQuotes(
    priceQuoteRequestPublisher: ActorRef)
    extends Actor {
    val quoterId = self.path.name

    priceQuoteRequestPublisher !
        SubscribeToPriceQuoteRequests(quoterId, self)

    def receive = {
        case rpq: RequestPriceQuote =>
            if (rpq.orderTotalRetailPrice < 2000.00) {
                val discount = discountPercentage(
                    rpq.orderTotalRetailPrice) *
                    rpq.retailPrice
                sender ! PriceQuote(
                    quoterId,
                    rpq.rfqId,
                    rpq.itemId,
                    rpq.retailPrice,
                    rpq.retailPrice - discount)
            } else {
                println(s"RockBottomOuterwearPriceQuotes: $rpq")
                ignoring: $rpq")
            }
    }
}

```

```

    }

    case message: Any =>
      println(s"RockBottomOuterwearPriceQuotes: unexpected: '$message'")
  }

  def discountPercentage(orderTotalRetailPrice: Double) = {
    if (orderTotalRetailPrice <= 100.00) 0.015
    else if (orderTotalRetailPrice <= 399.99) 0.02
    else if (orderTotalRetailPrice <= 499.99) 0.03
    else if (orderTotalRetailPrice <= 799.99) 0.04
    else if (orderTotalRetailPrice <= 999.99) 0.05
    else if (orderTotalRetailPrice <= 2999.99) 0.06
    else if (orderTotalRetailPrice <= 4999.99) 0.07
    else if (orderTotalRetailPrice <= 5999.99) 0.075
    else 0.08
  }
}

```

运行该程序会获得下列结果：

```

OrderProcessor: 123 item: 1 to: rockBottomOuterwear
OrderProcessor: 123 item: 2 to: rockBottomOuterwear
OrderProcessor: 123 item: 3 to: rockBottomOuterwear
OrderProcessor: 123 item: 1 to: highSierra
OrderProcessor: 123 item: 2 to: highSierra
OrderProcessor: 123 item: 3 to: highSierra
OrderProcessor: 123 item: 1 to: pinnacleGear
OrderProcessor: 123 item: 2 to: pinnacleGear
OrderProcessor: 123 item: 3 to: pinnacleGear
OrderProcessor: 123 item: 1 to: budgetHikers
OrderProcessor: 123 item: 2 to: budgetHikers
OrderProcessor: 123 item: 3 to: budgetHikers
OrderProcessor: 123 item: 1 to: mountainAscent
OrderProcessor: 123 item: 2 to: mountainAscent
OrderProcessor: 123 item: 3 to: mountainAscent
...
BudgetHikersPriceQuotes: ignoring: RequestPriceQuote(125,4,39.99,1114.88)
OrderProcessor: 129 item: 11 to: mountainAscent
BudgetHikersPriceQuotes: ignoring: RequestPriceQuote(125,5,199.95,1114.88)
OrderProcessor: 135 item: 12 to: rockBottomOuterwear
BudgetHikersPriceQuotes: ignoring: RequestPriceQuote(

```

```

125,6,149.95,1114.88)
OrderProcessor: 135 item: 13 to: rockBottomOuterwear
BudgetHikersPriceQuotes: ignoring: RequestPriceQuote(
125,7,724.99,1114.88)
OrderProcessor: 135 item: 14 to: rockBottomOuterwear
BudgetHikersPriceQuotes: ignoring: RequestPriceQuote(
129,8,119.99,1348.44)
OrderProcessor: 135 item: 12 to: highSierra
BudgetHikersPriceQuotes: ignoring: RequestPriceQuote(
129,9,499.95,1348.44)
OrderProcessor: 135 item: 13 to: highSierra
BudgetHikersPriceQuotes: ignoring: RequestPriceQuote(
129,10,519.0,1348.44)
OrderProcessor: 135 item: 14 to: highSierra
OrderProcessor: 135 item: 12 to: pinnacleGear
...
PriceQuoteAggregator: fulfilled price quote:
  PriceQuoteFulfilled
OrderProcessor: received: PriceQuote(highSierra,125,
4,39.99,38.3904)
OrderProcessor: received: PriceQuote(highSierra,125,
5,199.95,191.952)
OrderProcessor: received: PriceQuote(highSierra,125,
6,149.95,143.952)
...
OrderProcessor: received:
  BestPriceQuotation(140,Vector(PriceQuote(mountainAscent,
140,15,1295.5,1233.96375),
  PriceQuote(mountainAscent,140,18,249.95,238.077375),
  PriceQuote(mountainAscent,140,17,599.99,571.4904750000001),
  PriceQuote(mountainAscent,140,16,9.5,9.04875),
  PriceQuote(mountainAscent,140,19,789.99,752.465475)))
OrderProcessor: received:
  BestPriceQuotation(125,Vector(PriceQuote(rockBottomOuterwear,
125,5,199.95,187.953),
  PriceQuote(rockBottomOuterwear,125,7,724.99,681.4906),
  PriceQuote(rockBottomOuterwear,125,4,39.99,37.5906),
  PriceQuote(rockBottomOuterwear,125,6,149.95,140.953)))
OrderProcessor: received: BestPriceQuotation(129,
Vector(PriceQuote(rockBottomOuterwear,129,8,119.99,
112.7906), PriceQuote(rockBottomOuterwear,129,11,
209.5,196.93), PriceQuote(rockBottomOuterwear,129,
9,499.95,469.953), PriceQuote(rockBottomOuterwear,129,
10,519.0,487.86)))
Scatter-Gather: is completed.

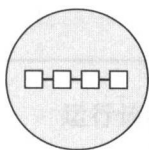
```

BudgetHikersPriceQuotes 和 RockBottomOuterwearPriceQuotes 对

象能够将大多数报价分成两部分。而 MountainAscentPriceQuotes 对象能够获得一条 RequestForQuotation 消息，因为 BudgetHikersPriceQuotes 和 RockBottomOuterwearPriceQuotes 对象分别拒绝获取 1,000 美元和 2,000 美元以上的报价，而且第 140 条 RequestForQuotation 消息的报价高于 2,900 美元。通过忽略指定范围中的报价，该示例程序会遇到超时情况，并且会与多个卖家交易。

分散—聚集模式是由接收者列表和聚合器或者发布—订阅通道构成的路由模式，通过这种模式可以实现较大的组合消息处理器。为什么要使用多个提供更精细路由功能的组件，构建较大的路由器呢？这样做有助于将组合消息处理器用作管道和过滤器模式中的独立过滤器。

传送名单



当某个大型的业务过程从逻辑方面看仅执行一个操作，但从物理方面看含有一系列处理步骤时，就应使用传送名单。这能够实现 SOA（面向服务架构的）服务组合。在传送名单处理过程中，每个步骤都是由独立的 Actor 对象处理的。下面的示例介绍了客户注册过程（请参阅 *Enterprise Integration Patterns* 一书），该过程含有下列步骤：

1. 创建新客户。
2. 记录客户的联系信息。
3. 为客户申请服务计划。
4. 检查新客户的信用情况。

下面让我们先看看这个示例程序（RoutingSlip.scala）的开头和构成注册消息数据体的值对象。

```
package co.vaughnvernon.reactiveenterprise.routingslip

import akka.actor._
import co.vaughnvernon.reactiveenterprise._

case class CustomerInformation(
```

```

    val name: String,
    val federalTaxId: String)

case class ContactInformation(
    val postalAddress: PostalAddress,
    val telephone: Telephone)

case class PostalAddress(
    val address1: String,
    val address2: String,
    val city: String,
    val state: String,
    val zipCode: String)

case class Telephone(val number: String)

case class ServiceOption(
    val id: String,
    val description: String)

case class RegistrationData(
    val customerInformation: CustomerInformation,
    val contactInformation: ContactInformation,
    val serviceOption: ServiceOption)

```

上面的代码构成了不可变的值对象 `RegistrationData` [IDDD]。下面是实现传送名单的 `RegistrationProcess` 对象，该对象由多个独立的 `ProcessStep` 实例构成。

```

case class ProcessStep(
    val name: String,
    val processor: ActorRef)

case class RegistrationProcess(
    val processId: String,
    val processSteps: Seq[ProcessStep],
    val currentStep: Int) {

    def this(
        processId: String,
        processSteps: Seq[ProcessStep]) {
        this(processId, processSteps, 0)
    }

    def isCompleted: Boolean = {

```

```

    currentStep >= processSteps.size
  }

  def nextStep(): ProcessStep = {
    if (isCompleted) {
      throw new IllegalStateException(
        "Process had already completed.")
    }

    processSteps(currentStep)
  }

  def stepCompleted(): RegistrationProcess = {
    new RegistrationProcess(
      processId,
      processSteps,
      currentStep + 1)
  }
}

```

每个 `ProcessStep` 实例都拥有名称，和指向执行处理步骤的 `Actor` 对象的引用。`RegistrationProcess` 对象负责标记已经完成的处理步骤，确定整个处理过程是否已经完成，和检索需要处理的下一个步骤。在本例中，处理过程中的所有 `Actor` 对象都会收到类型相同的消息：`RegisterCustomer`。

```

case class RegisterCustomer(
  val registrationData: RegistrationData,
  val registrationProcess: RegistrationProcess) {

  def advance(): Unit = {
    val advancedProcess =
      registrationProcess.stepCompleted
    if (!advancedProcess.isCompleted) {
      advancedProcess.nextStep().processor !
        RegisterCustomer(
          registrationData,
          advancedProcess)
    }
    RoutingSlip.completedStep()
  }
}

```

当 `Actor` 对象收到 `RegisterCustomer` 消息（代表注册的客户）时，会使用该消息包含的 `RegistrationData` 字段（代表注册数据）中的部分数据，执

行该 Actor 对象在处理过程中负责完成的步骤（请参阅前面介绍的处理步骤）。独立处理步骤中的倒数第二个操作，是使 RegistrationProcess 对象切换到下一个步骤，并将该步骤分派给负责处理它的 Actor 对象（稍后会介绍最后一个操作）。每个 Actor 对象都使用收到的 RegisterCustomer 消息中的 advance() 方法做到这一点。

```
val registrationData =
    registerCustomer.registrationData
...
registerCustomer.advance()
...
```

这使 RegisterCustomer 消息成为一种封装器。advance() 方法会创建新的不可变的 RegisterCustomer 消息，但是会通过增加 currentStep 索引的值，使新消息指向下一个需要处理的步骤。

本示例应用程序 (RoutingSlip 对象)，使用 ProcessStep 实例构成 RegistrationProcess 对象，并将 RegistrationProcess 对象中 currentStep 字段的值初始化为 0（指向第一个处理步骤）。

```
object RoutingSlip extends CompletableApp(4) {
    val processId = java.util.UUID.randomUUID().toString

    val step1 = ProcessStep(
        "create_customer",
        ServiceRegistry.customerVault(
            system,
            processId))

    val step2 = ProcessStep(
        "set_up_contact_info",
        ServiceRegistry.contactKeeper(
            system,
            processId))

    val step3 = ProcessStep(
        "select_service_plan",
        ServiceRegistry.servicePlanner(
            system,
            processId))

    val step4 = ProcessStep(
        "check_credit",
```

```

    ServiceRegistry.creditChecker(
        system,
        processId))

val registrationProcess =
    new RegistrationProcess(
        processId,
        Vector(
            step1,
            step2,
            step3,
            step4))

val registrationData =
    new RegistrationData(
        CustomerInformation(
            "ABC, Inc.", "123-45-6789"),
        ContactInformation(
            PostalAddress(
                "123 Main Street", "Suite 100",
                "Boulder", "CO", "80301"),
            Telephone("303-555-1212")),
        ServiceOption(
            "99-1203",
            "A description of 99-1203."))

val registerCustomer =
    RegisterCustomer(
        registrationData,
        registrationProcess)

registrationProcess
    .nextStep
    .processor ! registerCustomer

awaitCompletion
println("RoutingSlip: is completed.")
}

```

为了查找负责处理每个步骤的 Actor 对象，RoutingSlip 对象使用了 ServiceRegistry 对象。结果，每当需要查找 Actor 对象时，ServiceRegistry 对象就会创建一个新的 Actor 实例。重用同一个 Actor 对象（为每个处理步骤仅创建一个 Actor 对象）也很容易实现，但如果极少出现新注册的客户，就不必这样做。

```

object ServiceRegistry {
    def contactKeeper(

```

```

    system: ActorSystem,
    id: String) = {
  system.actorOf(
    Props[ContactKeeper],
    "contactKeeper-" + id)
}

def creditChecker(
  system: ActorSystem,
  id: String) = {
  system.actorOf(Props[CreditChecker],
    "creditChecker-" + id)
}

def customerVault(
  system: ActorSystem,
  id: String) = {
  system.actorOf(Props[CustomerVault],
    "customerVault-" + id)
}

def servicePlanner(
  system: ActorSystem,
  id: String) = {
  system.actorOf(Props[ServicePlanner],
    "servicePlanner-" + id)
}

```

还需要清理这些负责处理步骤的 Actor 对象，每个 Actor 对象都能够轻松完成这项任务。

```

class CreditChecker extends Actor {
  def receive = {
    case registerCustomer: RegisterCustomer =>
      val federalTaxId =
        registerCustomer.registrationData
          .customerInformation.federalTaxId

      println(s"CreditChecker: handling register
customer to perform credit check: $federalTaxId")

      registerCustomer.advance()

      context.stop(self)
    case message: Any =>

```

```

        println(s"CreditChecker: unexpected: $message")
    }
}

class ContactKeeper extends Actor {
    def receive = {
        case registerCustomer: RegisterCustomer =>
            val contactInfo =
                registerCustomer.registrationData
                    .contactInformation

            println(s"ContactKeeper: handling register
customer to keep contact information: $contactInfo")

            registerCustomer.advance()

            context.stop(self)
        case message: Any =>
            println(s"ContactKeeper: unexpected: $message")
    }
}

class CustomerVault extends Actor {
    def receive = {
        case registerCustomer: RegisterCustomer =>
            val customerInformation =
                registerCustomer.registrationData
                    .customerInformation

            println(s"CustomerVault: handling register
customer to create a new customer: $customerInformation")

            registerCustomer.advance()

            context.stop(self)
        case message: Any =>
            println(s"CustomerVault: unexpected: $message")
    }
}

class ServicePlanner extends Actor {
    def receive = {
        case registerCustomer: RegisterCustomer =>
            val serviceOption =
                registerCustomer
                    .registrationData.serviceOption
    }
}

```



```
println(s"ServicePlanner: handling register
customer to plan a new customer service: $serviceOption")

registerCustomer.advance()

context.stop(self)
case message: Any =>
  println(s"ServicePlanner: unexpected: $message")
}
```

一旦 Actor 对象处理完指定的业务逻辑，Actor 对象就会命令 RegisterCustomer 对象切换到下一个步骤，然后使自己停止运行。

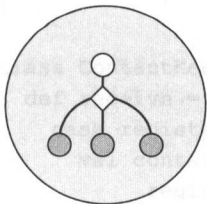
```
case registerCustomer: RegisterCustomer =>
  ...
  registerCustomer.advance()
  context.stop(self)
```

这个程序会输出下面的结果：

```
CustomerVault: handling register customer to create
a new customer:
  CustomerInformation(ABC, Inc.,123-45-6789)
ContactKeeper: handling register customer to keep
contact information:
  ContactInformation(
    PostalAddress(123 Main Street,Suite 100,Boulder,
CO,80301),
    Telephone(303-555-1212))
ServicePlanner: handling register customer to plan a
new customer service:
  ServiceOption(99-1203,A description of 99-1203.)
CreditChecker: handling register customer to perform
credit check: 123-45-6789
RoutingSlip: is completed.
```

可以使用与本例不同的方式组合 RegistrationProcess 对象，如调整它们的执行次序。而且，还可以添加处理步骤或在已存在的处理步骤之间插入处理步骤。这些传送名单的实现代码仍旧可以正常运行，因为其处理步骤的次序是使用 Scala 的 Seq 序列设置的，在本例中该类序列被实现为 Vector 集合。

处理过程管理器



即使在设计程序时不知道有哪些必要步骤和排列处理步骤的顺序,使用处理过程管理器也可以为消息提供经过多个处理步骤的路由。传送名单就是一种处理步骤管理器,但它只能处理一系列固定的线性处理步骤。在不使用条件分支或循环的情况中,可选择使用较简单的传送名单路由器。然而,当遇到如图 7.10 所示的情况时,就应该使用功能更强的处理过程管理器。注意,分派给各家银行的处理步骤都是以并发方式发送的,因而都使用了相同的处理步骤编号 2。

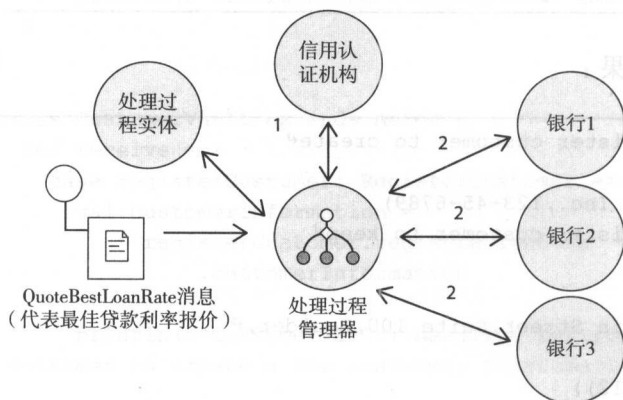


图 7.10 如果在设计程序时不知道处理过程中有哪些必要步骤和排列处理步骤的顺序,可使用处理过程管理器。每家银行的并行处理步骤使用了相同的次序编号 2。

实现处理过程管理器的方式主要有两个。一种方式是创建一门领域专用语言(DSL)和一个解释程序。这门 DSL 本身应该是高级编程语言,以便使你能够描述各种业务处理过程中的规则。业务流程执行语言(BPEL)就是一种 DSL,其中含有用于阅读、解释和管理该语言定义的业务处理过程的多种工具。在使用这种方式创建处理过程管理器时,处理过程管理器本身无法成为处理过程定义的核心组成部分,因为必须使解释器能够管理范围较广的业务处理过程。这种处理过程管理器不是专门应用于具体业务领域的。换言之,这种处理过程管理器本身具有较高的可重用性,与高级编程语言的编译器/解释器类似。

我不是要在此处介绍创建这类具有解释功能的处理过程管理器的方式。我要介绍一个通过第二种方式实现的处理过程管理器。这个处理过程管理器专门用于处理贷款利率报价请求，因此它是一个专门用于具体业务领域的处理过程管理器。该处理过程管理器必须从多家银行获取贷款利率报价，然后在这些贷款利率报价中找出最佳贷款利率报价。流程控制规则是直接专业处理过程管理器中实现的，使用保存处理过程状态的独立处理过程实体有助于增强处理过程管理器的功能，而且使用独立的处理过程实体还可以根据处理过程的状态提供处理过程转换方向。表 7.1 介绍了本例使用的 Actor 对象。我建议你在设计自己的 Actor 应用程序时，也使用这种方式为处理过程管理器建模。这是一种根据 Actor 对象的主要角色创建模型的方式，该建模方式很适合由领域驱动的设计方式。

表 7.1 贷款利率报价处理过程管理器的组件

Actor 对象	描述
LoanBroker	这是一个被实现为消息经纪人的处理过程管理器。该对象接收由 QuoteBestLoanRate 消息代表的原始报价请求。该对象最终会生成 BestLoanRateQuoted 或 BestLoanRateDenied 消息
LoanRateQuote	该对象是处理过程实体，处理过程管理器使用该对象保存处理过程状态，并通过它获得处理过程转换方向。当处理过程管理器收到 QuoteBestLoanRate 消息时，就会创建一个处理过程实体，该实体会在整个贷款利率报价处理过程中被使用。这个实体会收集所有银行提供的贷款利率报价，当获得了所有预期的报价后，该实体就会将这些报价中的最佳报价发送给 LoanBroker 对象
CreditBureau	该对象会为请求获取贷款利率报价的各方提供信用评分信息。该 Actor 对象接收 CheckCredit 命令消息，并会回复 CreditChecked 事件消息
Bank1、Bank2、Bank3	这三个对象的类型都是 Bank，但本例的应用程序使用 Bank 类创建了这 3 个代表银行的 Actor 实例。通过对这 3 个对象使用不同的配置和随机乘法器，每个对象都能够为 QuoteBestLoanRate 请求消息，提供不同的贷款利率报价

尽管我通常不喜欢将控制点集中到一起，但处理过程管理器趋向于将控制点集中起来。然而，这并不意味着无法分散控制点。例如，通过使用发布—订阅通道示例中介绍的 DistributedPubSubMediator 类，可以实现分布式的处理过程管理器。但下面的示例使用的是以集中 Actor 对象为基础的设计。

要设计处理过程管理器，应先考虑容纳它的应用程序。

```
object ProcessManagerDriver extends CompletableApp(5) {
```

```

val creditBureau =
  system.actorOf(
    Props[CreditBureau],
    "creditBureau")

val bank1 =
  system.actorOf(
    Props(classOf[Bank], "bank1", 2.75, 0.30),
    "bank1")
val bank2 =
  system.actorOf(
    Props(classOf[Bank], "bank2", 2.73, 0.31),
    "bank2")
val bank3 =
  system.actorOf(
    Props(classOf[Bank], "bank3", 2.80, 0.29),
    "bank3")

val loanBroker = system.actorOf(
  Props(classOf[LoanBroker],
    creditBureau,
    Vector(bank1, bank2, bank3)),
  "loanBroker")
loanBroker ! QuoteBestLoanRate("111-11-1111", 100000, 84)

awaitCompletion
}

```

应用程序 `ProcessManagerDriver` 先创建了 Actor 对象 `CreditBureau`，然后创建了 3 个 `Bank` 类型的 Actor 对象。每个 `Bank` 类型的 Actor 对象都被赋予了不同的最优惠贷款利率和溢价贷款利率，这些数据用于计算各份贷款利率报价。最后一个被创建的 Actor 对象是处理过程管理器 (`LoanBroker`)，该对象会被赋予 Actor 对象 `CreditBureau` 的引用和所有 `Bank` 类型 Actor 对象的引用。然后，该应用程序将 `QuoteBestLoanRate` 消息发送给 `LoanBroker` 对象。这条消息会起到触发器的作用，使一个复杂的处理过程开始运行。从客户端方面看，这是一个简单的请求—回复处理过程。

`LoanBroker` 类扩展了抽象基类 `ProcessManager`。该基类本身扩展了 Actor 类，并且还提供了一些基本的、可重用的处理过程管理器操作。

```

case class ProcessStarted(
  processId: String,
  process: ActorRef)

```

```

case class ProcessStopped(
  processId: String,
  process: ActorRef)

abstract class ProcessManager extends Actor {
  private var processes = Map[String, ActorRef]()
  val log: LoggingAdapter = Logging.getLogger(context.system, self)

  def processOf(processId: String): ActorRef = {
    if (processes.contains(processId)) {
      processes(processId)
    } else {
      null
    }
  }

  def startProcess(processId: String, process: ActorRef) = {
    if (!processes.contains(processId)) {
      processes = processes + (processId -> process)
      self ! ProcessStarted(processId, process)
    }
  }

  def stopProcess(processId: String) = {
    if (processes.contains(processId)) {
      val process = processes(processId)
      processes = processes - processId
      self ! ProcessStopped(processId, process)
    }
  }
}

```

抽象基类 `ProcessManager` 用于管理每个处理过程的生命周期。当新的处理过程开始运行时，执行该处理过程的 `Actor` 对象会使用 `startProcess()` 函数，根据标识符保存新的处理过程实体。在这种实现方式中，除了将处理过程实体保存在内存中外，只能将它们保存在数据库中。`startProcess()` 函数会通过向自己发送 `ProcessStarted` 消息做出回应。当担任处理过程管理器的 `Actor` 对象需要与代表处理过程实体的 `Actor` 对象交互时，会将处理过程实体的唯一标识符发送给 `processOf()` 方法。当处理过程到达完成状态时，代表该处理过程的 `Actor` 对象会使用 `stopProcess()` 方法删除已保存的状态，并给自己发送 `ProcessStopped` 消息。通过上述两种生命周期消息，担任处理过程管理器的 `Actor` 对象可以执行特殊的操作，如命令 `Actor` 对象停止运行。

```

class LoanBroker(
  creditBureau: ActorRef,
  banks: Seq[ActorRef])
  extends ProcessManager {
  ...
  case message: ProcessStopped =>
    log.info(s"$message")
    context.stop(message.process) ...
}

```

在本例中，处理过程管理器由 `LoanBroker` 对象担任。`LoanBroker` 对象会接收生命周期消息 `ProcessStopped`，并使代表处理过程实体的 `Actor` 对象停止运行。

下面让我们看看 `LoanBroker` 对象的其余实现代码。下面先介绍 `LoanBroker` 对象外部协定中的消息类型。

```

case class QuoteBestLoanRate(
  taxId: String,
  amount: Integer,
  termInMonths: Integer)

case class BestLoanRateQuoted(
  bankId: String,
  loanRateQuoteId: String,
  taxId: String,
  amount: Integer,
  termInMonths: Integer,
  creditScore: Integer,
  interestRate: Double)

case class BestLoanRateDenied(
  loanRateQuoteId: String,
  taxId: String,
  amount: Integer,
  termInMonths: Integer,
  creditScore: Integer)

```

客户端会向 `Actor` 对象 `LoanBroker` 发送 `QuoteBestLoanRate` 命令消息（代表获取最佳报价的请求），并想要从 `LoanBroker` 对象获得回复的 `BestLoanRateQuoted` 事件消息。然而，在提供贷款利率报价时可能会出问题。如果某个请求获取贷款利率报价的人的贷款信用过低，那么 `LoanBroker` 对象就

会向该客户端回复 `BestLoanRateDenied` 事件消息（代表拒绝提供最佳贷款利率报价）。

下面是 `LoanBroker` 对象处理它收到的 `QuoteBestLoanRate` 命令消息的方式。

```
class LoanBroker(
  creditBureau: ActorRef,
  banks: Seq[ActorRef])
  extends ProcessManager {
  ...
  case message: QuoteBestLoanRate =>
    val loanRateQuoteId = LoanRateQuote.id

    log.info(s"$message for: $loanRateQuoteId")

    val loanRateQuote =
      LoanRateQuote(
        context.system,
        loanRateQuoteId,
        message.taxId,
        message.amount,
        message.termInMonths,
        self)

    startProcess(loanRateQuoteId, loanRateQuote)
  }
  ...
}
```

在开始运行每个处理过程时，本应用程序都会创建一个新的 `LoanRateQuote` 对象，该 `Actor` 对象代表处理过程实体，用于保存所有报价请求的状态。创建好该对象后，基于 `Actor` 的处理过程就开始运行了。此后处理过程就会像瀑布一样奔流直下。但是，你还必须通过观察处理 `ProcessStarted` 消息的对象和 `Actor` 对象 `LoanRateQuote`，查明处理过程管理器和被管理的处理过程之间的协作情况。

```
class LoanBroker(
  creditBureau: ActorRef,
  banks: Seq[ActorRef])
  extends ProcessManager {
  ...
  case message: ProcessStarted =>
    log.info(s"$message")
```

```

message.process ! StartLoanRateQuote(banks.size)
...
}

```

LoanBroker 对象将生命周期消息 ProcessStarted 用作向 LoanRateQuote 对象发送第一条业务处理消息 StartLoanRateQuote 的引子。StartLoanRateQuote 消息中含有提供贷款利率报价的银行的数量。LoanRateQuote 对象必须了解该信息，它还要起消息聚合器的作用，收集各家银行提供的所有报价。你可以像处理 LoanRateQuote 对象的其他配置一样，将提供报价的银行的数量传输为构造器参数。而且，通过 StartLoanRateQuote 消息发送提供报价的银行的数量，有助于更精确地在处理过程中合并指定数量的银行报价。

```

class LoanRateQuote(
  loanRateQuoteId: String,
  taxId: String,
  amount: Integer,
  termInMonths: Integer,
  loanBroker: ActorRef)
  extends Actor {

  var bankLoanRateQuotes = Vector[BankLoanRateQuote]()
  var creditRatingScore: Int = _
  var expectedLoanRateQuotes: Int = _

  def receive = {
    case message: StartLoanRateQuote =>
      expectedLoanRateQuotes =
        message.expectedLoanRateQuotes
      loanBroker !
        LoanRateQuoteStarted(
          loanRateQuoteId,
          taxId)
      ...
  }
}

```

一旦将 expectedLoanRateQuotes 字段（代表银行提供的贷款利率报价）的值记录到 LoanRateQuote 对象中，该对象就会对 LoanBroker 对象发送一条 LoanRateQuoteStarted 消息。收到这条消息后，LoanBroker 对象就知道它可以通过发送 CheckCredit 消息，向 CreditBureau 对象请求获取下一条信用评分记录了。这种协同的交互操作会贯穿整个处理过程的生命周期。为了彻底了解该处理过程，你需要跟踪在 LoanBroker、LoanRateQuote、CreditBureau 和多个 Bank 实例之间发送的消息。图 7.11 展示了这些发送消息的活动(大部分)。

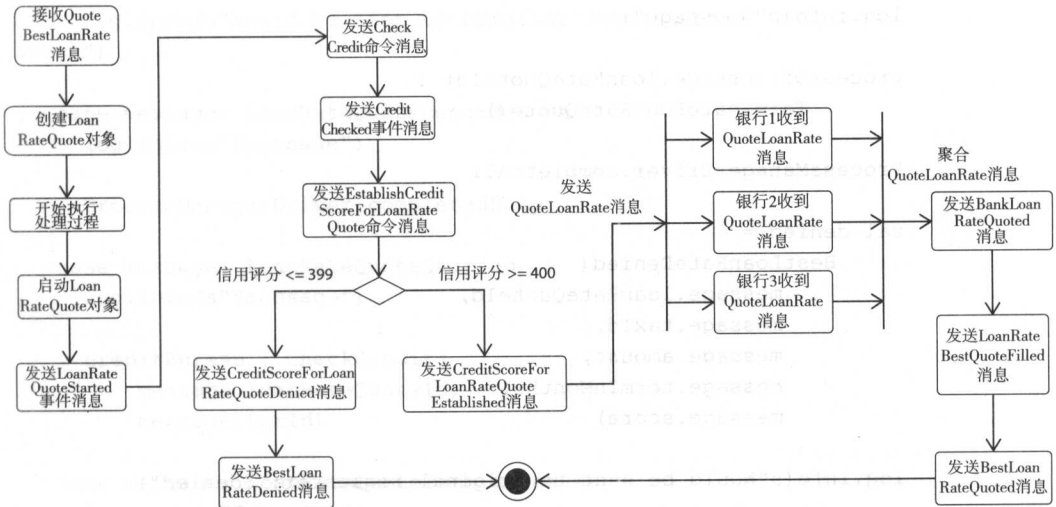


图 7.11 为了支持贷款利率报价处理过程，处理过程管理器所执行的操作。

下面是 LoanBroker 对象完整的实现代码：

```

class LoanBroker(
    creditBureau: ActorRef,
    banks: Seq[ActorRef])
    extends ProcessManager {

    def receive = {
        case message: BankLoanRateQuoted =>
            log.info(s"$message")

            processOf(message.loadQuoteReferenceId) !
                RecordLoanRateQuote(
                    message.bankId,
                    message.bankLoanRateQuoteId,
                    message.interestRate)

        case message: CreditChecked =>
            log.info(s"$message")

            processOf(message.creditProcessingReferenceId) !
                EstablishCreditScoreForLoanRateQuote(
                    message.creditProcessingReferenceId,
                    message.taxId,
                    message.score)

        case message: CreditScoreForLoanRateQuoteDenied =>
    
```

```

log.info(s"$message")

processOf(message.loanRateQuoteId) !
    TerminateLoanRateQuote()

ProcessManagerDriver.completeAll

val denied =
    BestLoanRateDenied(
        message.loanRateQuoteId,
        message.taxId,
        message.amount,
        message.termInMonths,
        message.score)

log.info(s"Would be sent to original requester: $denied")

case message: CreditScoreForLoanRateQuoteEstablished =>
    log.info(s"$message")

    banks map { bank =>
        bank ! QuoteLoanRate(
            message.loanRateQuoteId,
            message.taxId,
            message.score,
            message.amount,
            message.termInMonths)
    }

ProcessManagerDriver.completedStep

case message: LoanRateBestQuoteFilled =>
    log.info(s"$message")

    ProcessManagerDriver.completedStep

    stopProcess(message.loanRateQuoteId)

    val best = BestLoanRateQuoted(
        message.bestBankLoanRateQuote.bankId,
        message.loanRateQuoteId,
        message.taxId,
        message.amount,
        message.termInMonths,
        message.creditScore,
        message.bestBankLoanRateQuote.interestRate)

```

```
log.info(s"Would be sent to original requester:$best")
```

```
case message: LoanRateQuoteRecorded =>
  log.info(s"$message")
```

```
ProcessManagerDriver.completedStep
```

```
case message: LoanRateQuoteStarted =>
  log.info(s"$message")
```

```
creditBureau ! CheckCredit(
  message.loanRateQuoteId,
  message.taxId)
```

```
case message: LoanRateQuoteTerminated =>
  log.info(s"$message")
```

```
stopProcess(message.loanRateQuoteId)
```

```
case message: ProcessStarted =>
  log.info(s"$message")
```

```
message.process ! StartLoanRateQuote(banks.size)
```

```
case message: ProcessStopped =>
  log.info(s"$message")
```

```
context.stop(message.process)
```

```
case message: QuoteBestLoanRate =>
  val loanRateQuoteId = LoanRateQuote.id
```

```
log.info(s"$message for: $loanRateQuoteId")
```

```
val loanRateQuote =
  LoanRateQuote(
    context.system,
    loanRateQuoteId,
    message.taxId,
    message.amount,
    message.termInMonths,
    self)
```

```
startProcess(loanRateQuoteId, loanRateQuote)
```

```
}
```

```
}
```

下面是 LoanRateQuote 对象的实现代码，其中包含了它的协定消息和伴生对象：

```

case class StartLoanRateQuote(
    expectedLoanRateQuotes: Integer)

case class LoanRateQuoteStarted(
    loanRateQuoteId: String,
    taxId: String)

case class TerminateLoanRateQuote()

case class LoanRateQuoteTerminated(
    loanRateQuoteId: String,
    taxId: String)

case class EstablishCreditScoreForLoanRateQuote(
    loanRateQuoteId: String,
    taxId: String,
    score: Integer)

case class CreditScoreForLoanRateQuoteEstablished(
    loanRateQuoteId: String,
    taxId: String,
    score: Integer,
    amount: Integer,
    termInMonths: Integer)

case class CreditScoreForLoanRateQuoteDenied(
    loanRateQuoteId: String,
    taxId: String,
    amount: Integer,
    termInMonths: Integer,
    score: Integer)

case class RecordLoanRateQuote(
    bankId: String,
    bankLoanRateQuoteId: String,
    interestRate: Double)

case class LoanRateQuoteRecorded(
    loanRateQuoteId: String,
    taxId: String,
    bankLoanRateQuote: BankLoanRateQuote)

```

```

case class LoanRateBestQuoteFilled(
  loanRateQuoteId: String,
  taxId: String,
  amount: Integer,
  termInMonths: Integer,
  creditScore: Integer,
  bestBankLoanRateQuote: BankLoanRateQuote)

case class BankLoanRateQuote(
  bankId: String,
  bankLoanRateQuoteId: String,
  interestRate: Double)

object LoanRateQuote {
  val randomLoanRateQuoteId = new Random()

  def apply(
    system: ActorSystem,
    loanRateQuoteId: String,
    taxId: String,
    amount: Integer,
    termInMonths: Integer,
    loanBroker: ActorRef): ActorRef = {

    val loanRateQuote =
      system.actorOf(
        Props(
          classOf[LoanRateQuote],
          loanRateQuoteId, taxId,
          amount, termInMonths, loanBroker),
        "loanRateQuote-" + loanRateQuoteId)

    loanRateQuote
  }

  def id() = {
    randomLoanRateQuoteId.nextInt(1000).toString
  }
}

class LoanRateQuote(
  loanRateQuoteId: String,
  taxId: String,
  amount: Integer,
  termInMonths: Integer,
  loanBroker: ActorRef)

```

```

extends Actor {

  var bankLoanRateQuotes = Vector[BankLoanRateQuote]()
  var creditRatingScore: Int = _
  var expectedLoanRateQuotes: Int = _

  private def bestBankLoanRateQuote() = {
    var best = bankLoanRateQuotes(0)

    bankLoanRateQuotes map { bankLoanRateQuote =>
      if (best.interestRate >
          bankLoanRateQuote.interestRate) {
        best = bankLoanRateQuote
      }
    }

    best
  }

  private def quotableCreditScore(
    score: Integer): Boolean = {
    score > 399
  }

  def receive = {
    case message: StartLoanRateQuote =>
      expectedLoanRateQuotes =
        message.expectedLoanRateQuotes
      loanBroker !
        LoanRateQuoteStarted(
          loanRateQuoteId,
          taxId)

    case message: EstablishCreditScoreForLoanRateQuote =>
      creditRatingScore = message.score
      if (quotableCreditScore(creditRatingScore))
        loanBroker !
          CreditScoreForLoanRateQuoteEstablished(
            loanRateQuoteId,
            taxId,
            creditRatingScore,
            amount,
            termInMonths)
      else
        loanBroker !
          CreditScoreForLoanRateQuoteDenied(

```



```

        loanRateQuoteId,
        taxId,
        amount,
        termInMonths,
        creditRatingScore)

case message: RecordLoanRateQuote =>
    val bankLoanRateQuote =
        BankLoanRateQuote(
            message.bankId,
            message.bankLoanRateQuoteId,
            message.interestRate)
    bankLoanRateQuotes =
        bankLoanRateQuotes :+ bankLoanRateQuote
    loanBroker !
        LoanRateQuoteRecorded(
            loanRateQuoteId,
            taxId,
            bankLoanRateQuote)

    if (bankLoanRateQuotes.size >=
        expectedLoanRateQuotes)
        loanBroker !
            LoanRateBestQuoteFilled(
                loanRateQuoteId,
                taxId,
                amount,
                termInMonths,
                creditRatingScore,
                bestBankLoanRateQuote)

case message: TerminateLoanRateQuote =>
    loanBroker !
        LoanRateQuoteTerminated(
            loanRateQuoteId,
            taxId)
}
}

```

请仔细观察 `quotableCreditScore()` 函数。该函数含有业务规则，该业务规则规定只有信用评分高于 399 的个人，才有资格获取贷款利率报价信息。对于信用评分低于等于 399 的个人，`LoanRateQuote` 对象会向 `LoanBroker` 对象发送 `CreditScoreForLoanRateQuoteDenied` 消息，进而使 `LoanBroker` 对象向代表这些人的客户端发送 `BestLoanRateDenied` 消息。

下面是 CreditBureau 对象的实现代码：

```

case class CheckCredit(
  creditProcessingReferenceId: String,
  taxId: String)

case class CreditChecked(
  creditProcessingReferenceId: String,
  taxId: String,
  score: Integer)

class CreditBureau extends Actor {
  val creditRanges = Vector(300, 400, 500, 600, 700)
  val randomCreditRangeGenerator = new Random()
  val randomCreditScoreGenerator = new Random()

  def receive = {
    case message: CheckCredit =>
      val range =
        creditRanges(
          randomCreditRangeGenerator.nextInt(5))
      val score =
        range
        + randomCreditScoreGenerator.nextInt(20)

      sender !
        CreditChecked(
          message.creditProcessingReferenceId,
          message.taxId,
          score)
  }
}

```

下面是 Bank 对象的实现代码：

```

case class QuoteLoanRate(
  loadQuoteReferenceId: String,
  taxId: String,
  creditScore: Integer,
  amount: Integer,
  termInMonths: Integer)

case class BankLoanRateQuoted(
  bankId: String,
  bankLoanRateQuoteId: String,

```

```

loadQuoteReferenceId: String,
taxId: String,
interestRate: Double)

class Bank(
  bankId: String,
  primeRate: Double,
  ratePremium: Double)
  extends Actor {

  val randomDiscount = new Random()
  val randomQuoteId = new Random()

  private def calculateInterestRate(
    amount: Double,
    months: Double,
    creditScore: Double): Double = {

    val creditScoreDiscount = creditScore / 100.0 / 10.0 -
      (randomDiscount.nextInt(5) * 0.05)
    primeRate + ratePremium + ((months / 12.0) / 10.0) -
      creditScoreDiscount
  }

  def receive = {
    case message: QuoteLoanRate =>
      val interestRate =
        calculateInterestRate(
          message.amount.toDouble,
          message.termInMonths.toDouble,
          message.creditScore.toDouble)

      sender ! BankLoanRateQuoted(
        bankId, randomQuoteId.nextInt(1000).toString,
        message.loadQuoteReferenceId, message.taxId, interestRate)
  }
}

```

CreditBureau 对象和 Bank 对象都是以提供随机结果的方式实现的。Bank 对象使用个人信用评分确定可以提供贷款利率折扣，信用评分越高贷款利率就越优惠。

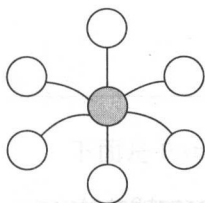
下面是使用能够获得最优惠贷款利率报价的信用评分，运行该示例程序获得的结果如下：

```

QuoteBestLoanRate(111-11-1111,100000,84) for: 151
ProcessStarted(151,Actor[akka://reactiveenterprise/user/loanRateQuote-151])
LoanRateQuoteStarted(151,111-11-1111)
CreditChecked(151,111-11-1111,610)
CreditScoreForLoanRateQuoteEstablished(151,111-11-1111,610,100000,84)
BankLoanRateQuoted(bank2,853,151,111-11-1111,3.33)
BankLoanRateQuoted(bank3,911,151,111-11-1111,3.23)
BankLoanRateQuoted(bank1,292,151,111-11-1111,3.34)
LoanRateQuoteRecorded(151,111-11-1111,BankLoanRateQuote(bank2,853,3.33))
LoanRateQuoteRecorded(151,111-11-1111,BankLoanRateQuote(bank3,911,3.23))
LoanRateQuoteRecorded(151,111-11-1111,BankLoanRateQuote(bank1,292,3.34))
LoanRateBestQuoteFilled(151,111-11-1111,100000,84,610,BankLoanRateQuote(bank3,911,3.23))
Would be sent to original requester: BestLoanRateQuoted(bank3,151,111-11-1111,100000,84,610,3.23)
ProcessStopped(151,Actor[akka://reactiveenterprise/user/loanRateQuote-151])

```

消息经纪人路由器



使用消息经纪人可以在消息流程中保持中央控制功能的情况下，降低消息接收者和消息发送者之间的耦合性。如图 7.12 所示，消息经纪人主要用于在独立的应用程序(但需要整合到一起的)之间传输消息。该图展示了一种消息经纪人结构，这种结构中有控制应用程序“子网”的本地消息经纪人，还有用于在不同应用程序子网之间传递消息的中央消息经纪人。

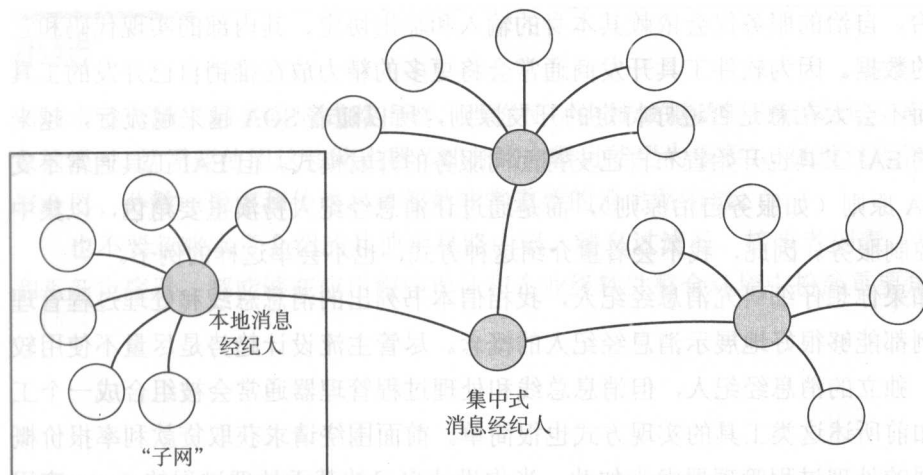


图 7.12 带有 3 个子网的集中式消息经纪人结构。

可以使用前面介绍的一种或多种消息路由模式实现消息经纪人。如 *Enterprise Integration Patterns* 一书所述，消息经纪人是一种架构模式，其中可以包含多种路由模式。因此，你可以根据具体情况，在消息经纪人中使用多种路由器设计模式。

消息经纪人需要解决的基本问题是，使由一个应用程序发送的消息能够经过消息经纪人被传输给另一个应用程序，以及将消息从一个应用程序使用的消息格式，转换成另一个应用程序使用的消息格式。如果所有消息都必须通过消息经纪人传输，那么消息经纪人不仅会变成传输消息的瓶颈，还会变成转换消息格式的瓶颈。增加大量必要的数据格式转换器，会增加性能开销。图 7.12 展示了 16 个不用种类的应用程序，这意味着要使这 16 个应用程序能够互相通信，需要添加 16^2 ，也就是 256 个数据格式转换器。¹

这些是许多企业级应用程序整合（EAI）工具都想要解决的问题。横向扩展消息经纪人中的消息接收器，是处理消息传输瓶颈的一种方式。EAI 工具通常会提供数据格式转换器，主要用作出售商品的商务应用程序的格式。除非你购买许可证，否则无法在自己的企业级应用程序中使用 EAI 工具提供的格式转换器。因此，下一章提供了转换大量消息格式的解决方案。你可以在自己的消息格式转换器中使用这些解决方案，这可能比使用 EAI 工具能取得更好的效果。

SOA 是在实现服务时推荐使用的架构和设计指导原则。其中的一条原则是服务之间通过消息进行通信，而不是通过直接调用。

¹ 这 16 个应用程序彼此都通信的情况好像不太容易出现，但出现这种情况的可能性确实存在。在细化投入了大量时间和精力业务方案时，就更不能排除这种可能性了。

务自治。自治的服务仅会依赖其本身的输入和输出协定、其内部的实现代码和它本身的数据。因为软件工具开发商通常会更多的精力放在推销自己开发的工具上，而不会太在意是否遵守特定的开发原则，所以随着 SOA 越来越流行，越来越多的 EAI 工具也开始宣布自己支持面向服务的开发模式。但 EAI 工具通常不支持 SOA 原则（如服务自治原则），而是通过让消息经纪人扮演重要角色，以集中方式控制服务。因此，我不会着重介绍这种方式，也不会举这样的例子。

如果你想仔细研究消息经纪人，我相信本书列出的消息总线和处理过程管理器示例都能够很好地展示消息经纪人的概念。尽管主流设计趋势是尽量不使用较大的、独立的消息经纪人，但消息总线和处理过程管理器通常会被组合成一个工具，如前所述这类工具的实现方式也很简单。前面围绕请求获取贷款利率报价概念实现的处理过程管理器尤为如此。当你设计自己的基于处理过程的 Actor 应用程序时，就应该使用这种方式创建处理过程管理器。

请思考消息经纪人有哪些地方与消息总线类似。可以将消息总线视为一种专用的消息通道，因为消息总线关注的焦点是，将在一条通道上传输的消息切换到另一条通道上进行传输的方式。消息路由器的职责就是确定使用哪条通道传输消息。换言之，被发送给消息总线的消息必定会被传送给已注册的、专门接收指定类型消息的 Actor 对象，这就要求消息总线必须提供维护和执行该路由规则的工具。因为消息总线和消息经纪人都以规范化消息模型为基础，所以这使得这两种路由器都属于基于内容的路由器。

为了处理数据格式转换问题，消息总线会使用通道适配器在应用程序和消息总线之间转换传输的消息的格式。该功能也与消息经纪人转换应用程序数据格式的功能类似或相同。

消息总线和消息经纪人之间也有不同之处。它们之间的一个差异是消息经纪人可以构成分层结构。即便如此，通过独立的、集中的方式使用消息总线，也可以将消息从一个子网传输到另一个子网。

在消息总线和消息经纪人差异极小的情况下，可以将消息总线视为一种消极的消息经纪人。这也许是一种公正的评价，而且也是最新的设计潮流离集中控制解决方案而去的原因之一。然而，虽然软件工具开发商使用消息经纪人、消息总线和处理步骤管理器等模式，创建较大的、昂贵的、运行速度缓慢的、复杂的 EAI 工具，但并不意味着在企业级软件解决方案中使用消息经纪人就会带来负面影响。实际上，应根据实际需要进行取舍。

小结

本章介绍了许多响应式路由器。当你使用由领域驱动的设计模式为 Actor 对象建模时,就需要使用这些路由器(部分路由器可能更为重要)。处理过程管理器、聚合器、分散—聚集和传送名单都是非常重要的消息路由器。

也不要忽略本章介绍的其他消息路由器。消息过滤器、接收者列表、分离器 and 重新定序器,都能够在应用程序设计和企业级软件整合环境中扮演重要角色。

第 8 章

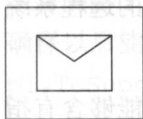
消息转换

第 4 章概要介绍过消息译码器。本章将详细介绍在应用程序设计和整合环境中使用的各种消息转换方式。

- **封装器**：在典型的中间件系统中，消息的头部具有独特的标准化属性。Actor 模型（尤其是 Akka 框架）不支持带头部的消息结构。然而，你可能需要使用类似消息头部的信封式结构封装某些消息。使用封装器可以实现一些特定的应用程序功能，如通道适配器和用于在请求—回复通信模式中分发回复消息的消息路由器。
- **内容丰富器**：有时 Actor 系统需要向整合器发送消息，其内部的消息类型可能因不够丰富，而无法被其他系统处理。可使用内容丰富器增强指定的消息，使之能够被特定的数据类型通道处理。
- **内容过滤器**：内容过滤器执行与内容丰富器相反的操作，从消息中移除指定的信息。它与消息过滤器不同，消息过滤器会过滤掉整条消息。更确切地说，内容过滤器用于从消息中去除需要保密的内容。
- **存放证**：当需要将一条组合消息分割成多个较小部分，并且还需要根据需求提供访问这些部分的操作时，可使用存放证。
- **标准化器**：标准化器是一种没有任何限制的转换器。当系统收到它不支持的消息类型时，可使用标准化器将这些消息转换成系统支持的消息类型。
- **规范化消息模型**：当需要将多个应用程序整合到一起时，可使用这种转换方式。例如，如果 5 至 7 个（或更多的）应用程序需要彼此发送消息，使用规范化消息模型可以创建所有应用程序都能够使用的通用消息。

尽管这些转换方式中的某些方式看起来不像用于转换消息的方式，但它们确实都是用于转换消息的。这些消息转换方式都脱胎自消息译码器。

封装器



前面介绍过在 Actor 模型中使用返回地址的几种方式，这使接收消息的 Actor 对象能够对消息发送者（或消息转发者）之外的其他 Actor 对象做出回复。然而，有时如果消息本身能够支持这种功能，会取得很好的效果。通过使消息包含封装器，可以做到这一点。

当传输过程中含有多个传输层次，或者当必须将消息隧道化才能在生成它的网络之外的其他网络中传输它时，就应该使用封装器。通常，Actor 系统（如 Akka 框架）生成的消息不必因这些缘由进行封装，而且 Actor 系统本身也可能提供专用的传输封装结构。但是，特定系统中的 Actor 对象还可能由于其他原因使用封装结构，如图 8.1 所示。

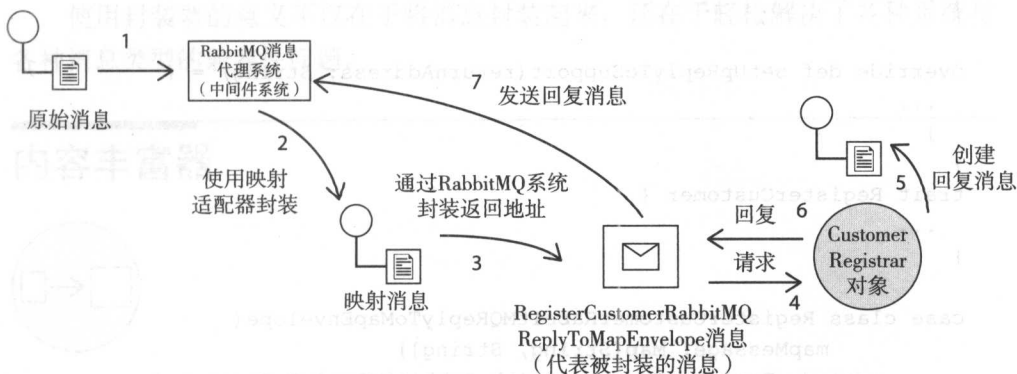


图 8.1 当需要保存中间件消息系统所需的回复地址时，可使用封装器。

例如，当原始消息进入系统时，所有 Actor 对象都不支持这种基础的消息类型。Actor 系统就可能封装这种不兼容的消息。通过封装消息可以实现下面的目的：

- 在封装器支持/扩展了 Actor 系统能够接受的消息类型协议的情况下 [GoF]，可将封装器用作原始消息与 Actor 系统支持的消息类型之间的适配器。
- 因为消息可能是由远程系统生成的，如果封装器提供了向生成消息的 Actor 对象做出回复的方式，那么就能够更轻松地实现这类回复操作。实际上生成消息的系统中的 Actor 对象，可能与本地系统中的 Actor 对象完全不同（例如，不是 Akka 框架中的 Actor 对象）。因此，对生成消息的

Actor 对象做出回复时，可能需要通过传输控制协议 /Internet 协议 (TCP/IP) 或其他传输和网络协议，使用一些面向消息的中间件。通过支持回复操作，封装器大幅度降低了本地系统中 Actor 对象与生成消息的远程系统中 Actor 对象进行交互的复杂度。

生成消息的系统可能会将键—值对映射发送为消息，这类消息中能够含有指明回复方式的元数据（如返回地址）。一旦消息到达基于 Actor 的本地系统，就会被封装然后发送给本地的 Actor 对象。注意，这个解决方案中使用了返回地址，但是在本例中它的作用是提供通过 RabbitMQ 系统向外部系统做回复的方式。

```

trait ReplyToSupport {
  def reply(message: Any) = { }
  def setUpReplyToSupport(returnAddress: String) = { }
}

trait RabbitMQReplyToSupport extends ReplyToSupport {
  override def reply(message: Any) = {
    ...
  }

  override def setUpReplyToSupport(returnAddress: String) = {
    ...
  }
}

trait RegisterCustomer {
  ...
}

case class RegisterCustomerRabbitMQReplyToMapEnvelope(
  mapMessage: Map[String, String])
  extends RegisterCustomer with RabbitMQReplyToSupport {
  this.setUpReplyToSupport(mapMessage("returnAddress"))
}

...

val mapMessage = receivedMessageAsMap(wireMessage)

val registerCustomer =
  RegisterCustomerRabbitMQReplyToMapEnvelope(mapMessage)
customerRegistrar ! registerCustomer

```

扩展的具体类必须像本例中的 RabbitMQReplyToSupport 特征一样，重

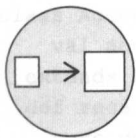
写 ReplyToSupport 抽象特征的行为。RegisterCustomerRabbitMQReplyToMapEnvelope 类扩展了 RabbitMQReplyToSupport 特征后, 就能够使用这个封装器了。具体的封装器会在构建过程中使用 setUpReplyToSupport() 方法, 确保以后能够成功使用 reply() 方法。通过收到的映射消息, 可以为 setUpReplyToSupport() 方法提供生成消息的 Actor 对象支持的返回地址元数据。

当 CustomerRegistrar 对象需要向生成消息的 Actor 对象做回复时, 只需用封装的回复消息。

```
class CustomerRegistrar extends Actor {
  def receive = {
    case registerCustomer: RegisterCustomer =>
      registerCustomer.reply(CustomerRegistered(...))
    case _ =>
      ...
  }
}
```

使用封装器的意义不仅在于将消息封装起来, 还在于轻松解决了各种系统与各种消息类型的兼容性问题。

内容丰富器



使用封装器能够使外部系统发送的不兼容消息与本地 Actor 系统兼容。但是, 也可能出现反过来的情况。由于本地 Actor 系统发送的消息所含内容不如外部系统的消息所含内容丰富, 导致本地 Actor 系统发送的消息与外部系统不兼容。更严重的情况是, 外部系统无法收到它需要的额外信息。使用内容丰富器可以解决这个问题。

Enterprise Integration Patterns 一书介绍了使用 Actor 对象提供的基础解决方案。在将消息发送给外部系统前, 该 Actor 对象会丰富消息的内容, 使消息含有所有必要内容, 如图 8.2 所示。

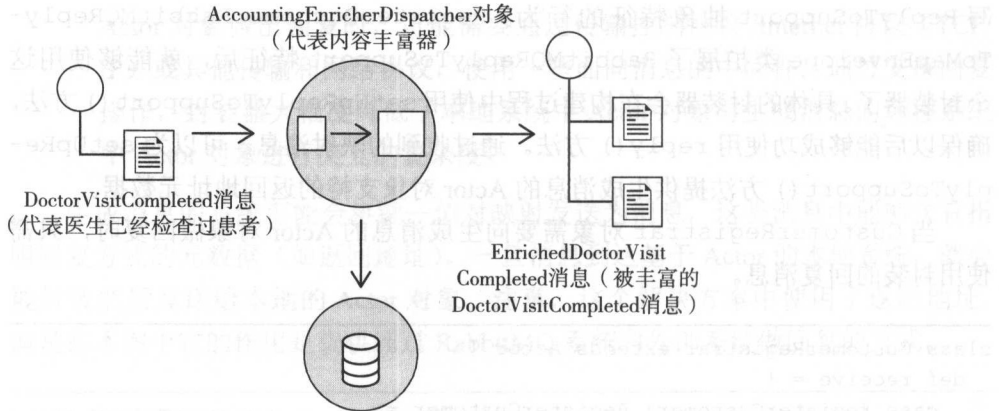


图 8.2 内容丰富器向 DoctorVisitCompleted 消息中添加了很多信息。额外的数据由数据库提供。

参照 *Enterprise Integration Patterns* 一书中介绍的示例，你可以使用所有必要信息丰富调度系统发送的消息，并将丰富后的消息发送给核算系统。

```
package co.vaughnvernon.reactiveenterprise.contentenricher
```

```
import akka.actor._
import co.vaughnvernon.reactiveenterprise._
import java.util.Date
```

```
case class DoctorVisitCompleted(
  val patientId: String,
  val firstName: String,
  val date: Date,
  val patientDetails: PatientDetails) {
  def this(patientId: String,
    firstName: String,
    date: Date) = {
    this(patientId, firstName, date,
      PatientDetails(null, null, null))
  }
  def carrier = patientDetails.carrier
  def lastName = patientDetails.lastName
  def socialSecurityNumber = patientDetails.socialSecurityNumber
}
```

```
case class PatientDetails(
  val lastName: String,
  val socialSecurityNumber: String,
```

```

    val carrier: String)

case class VisitCompleted(dispatcher: ActorRef)

object ContentEnricherDriver extends CompletableApp(3) {
    val accountingSystemDispatcher =
        system.actorOf(
            Props[AccountingSystemDispatcher],
            "accountingSystem")

    val accountingEnricherDispatcher =
        system.actorOf(
            Props(new AccountingEnricherDispatcher(
                accountingSystemDispatcher)),
            "accountingDispatcher")

    val scheduledDoctorVisit =
        system.actorOf(
            Props(new ScheduledDoctorVisit(
                "123456789", "John")),
            "scheduledVisit")

    scheduledDoctorVisit !
        VisitCompleted(accountingEnricherDispatcher)
    awaitCompletion
    println("ContentEnricher: is completed.")
}

class AccountingEnricherDispatcher(
    val accountingSystemDispatcher: ActorRef)
    extends Actor {
    def receive = {
        case doctorVisitCompleted: DoctorVisitCompleted =>
            println("AccountingEnricherDispatcher: "
                + "querying and forwarding.")
            // 查询用于执行丰富消息操作的患者信息 ...
            // ...
            val lastName = "Doe"
            val carrier = "Kaiser"
            val socialSecurityNumber = "111-22-3333"
            val enrichedDoctorVisitCompleted =
                DoctorVisitCompleted(
                    doctorVisitCompleted.patientId,
                    doctorVisitCompleted.firstName,
                    doctorVisitCompleted.date,
                    PatientDetails(
                        lastName,

```

```

        socialSecurityNumber,
        carrier))
    accountingSystemDispatcher forward
        enrichedDoctorVisitCompleted
    ContentEnricher.completedStep
case _ =>
    println("AccountingEnricherDispatcher: unexpected")
}
}

class AccountingSystemDispatcher extends Actor {
    def receive = {
        case doctorVisitCompleted: DoctorVisitCompleted =>
            println("AccountingSystemDispatcher: "
                + "sending to Accounting System...")
            ContentEnricher.completedStep
        case _ =>
            println("AccountingSystemDispatcher: unexpected")
    }
}

class ScheduledDoctorVisit(
    val patientId: String, val firstName: String)
    extends Actor {

    var completedOn: Date = _

    def receive = {
        case visitCompleted: VisitCompleted =>
            println("ScheduledDoctorVisit: completing visit.")
            completedOn = new Date()
            visitCompleted.dispatcher !
                new DoctorVisitCompleted(
                    patientId,
                    firstName,
                    completedOn)
            ContentEnricher.completedStep
        case _ =>
            println("ScheduledDoctorVisit: unexpected")
    }
}

```

运行该程序会得到下面的输出结果：

```

ScheduledDoctorVisit: completing visit.
AccountingEnricherDispatcher: querying and forwarding.

```



```
AccountingSystemDispatcher: sending to Accounting System...
ContentEnricher: is completed.
```

ScheduledDoctorVisit 对象被赋予了返回地址（即发送 DoctorVisitCompleted 消息的 Actor 对象的地址）。如果该消息不必进行丰富，那么就会由 AccountingSystemDispatcher 对象发送。然而，因为在将该消息发送给核算系统前必须丰富该消息，所以 CompleteVisit 消息中含有的是 AccountingEnricherDispatcher 对象的 ActorRef 引用。

可以让 AccountingEnricherDispatcher 对象直接将 DoctorVisitCompleted 消息发送给核算系统。但是，规定这项任务只能由 AccountingSystemDispatcher 对象完成会更好。因此，应将 AccountingEnricherDispatcher 对象编写为，将 DoctorVisitCompleted 消息转发给 AccountingSystemDispatcher 对象。

不可变的 DoctorVisitCompleted 消息

将 DoctorVisitCompleted 消息设计为不可变数据是很好的思路。领域事件不应该改变 [IDDD]，而 DoctorVisitCompleted 是一种事件消息。DoctorVisitCompleted 消息是通过两个构造器创建的，一个构造器仅接收本地属性，另一个构造器接收完整的 PatientDetails 对象（代表患者信息）。

这使 ScheduledDoctorVisit 对象能够发送与本地系统兼容的不可变消息。当 AccountingEnricherDispatcher 对象检索到额外的患者信息时，就会创建不可变的 DoctorVisitCompleted 实例。

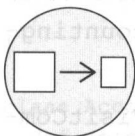
是否应在本地系统中创建 AccountingEnricherDispatcher 对象

是否应将 AccountingEnricherDispatcher 对象部署在调度系统中呢？你可以根据自己的喜好做决定。显然，*Enterprise Integration Patterns* 一书倾向于将该 Actor 对象放在调度系统外，因为它提出调度系统不应依赖客户关怀系统。因此，应该将 AccountingEnricherDispatcher 设置为远程 Actor 对象。另一方面，你可能会将 AccountingEnricherDispatcher 视为能够部署在任何实用系统中的独立 Actor 对象。在考虑竞争力的情况下，最终选定的方案会在很大程度上由特定系统部署 Actor 对象的能力决定。

但另一个因素也会影响该决策。如果整合调度系统和客户关怀系统的工作较复杂，而且调度系统无法支持这种重量级的整合工作，那么就必须将 AccountingEnricherDispatcher 对象移到调度系统之外。但是，如果能够通

过简化整合处理过程，降低客户关怀系统的限制，就能够在简化方面大做文章。在许多情况中，通过链接（URI）提供简单的 RESTful 资源，就能够简化搜索操作。

内容过滤器



在介绍封装器时，已经介绍过怎样处理外部消息与本地 Actor 系统不兼容的情况。使用封装器可以使外部消息适配本地系统，也可以使本地消息适配外部系统。这能够在外部消息系统和消息之间实现无缝结合。此外，前面还介绍了内容丰富器，使用内容丰富器可以在发送消息前，在消息中添加必要信息，以便使外部系统能够处理该消息。

然而，有时消息会含有过多信息或者含有难以处理的信息。请看下面的情况：

- 当执行本地应用程序专属的查询操作时，可以轻松从数据库获取数据。但是，这些丰富的数据可能过于敏感，以至于无法发送给外部系统。
- 查询到的本地应用程序数据可能过于庞大，因而无法通过网络发送，而且大多数消费者（甚至本地 Actor 对象）都不会用到其中的大多数数据。
- 数据的数量多而且结构也过于复杂，使某些本地 Actor 对象和远程系统都难以处理它们。

在遇到这些情况时，可使用内容过滤器。使用一个过滤器就足以简化和减少指定消息中包含的信息。另一方面，可能还需要使用分离器将较大的数据结构拆分成多个较小的消息类型。

注意，使用内容过滤器的目的与使用封装器的目的不同。使用内容过滤器不仅要较大的、较复杂的数据结构转换为消费者能够处理的数据格式，还要将消息含有的内容减少到消费者易于接收的程度。

内容过滤器与管道和过滤器

内容过滤器不必与管道和过滤器结构中的过滤器相同。在定义管道和过滤器模式中的过滤器时，过滤器仅是一种不改变消息内容的处理器。但是，根据具体情况，也可以使用内容丰富器或内容过滤器替换它。

当然，内容过滤器应该位于发送系统和 / 或接收系统中。如果认证操作是需

要考虑的重点，那么就应该将内容过滤器放在发送者系统中。如果消息包含的数据量很大，尽管需要进行认证，但发送系统还需要考虑网络优化问题。

然而，在整合过程中处理复杂的第三方消息（甚至能够被公开）时，网络优化的重要性低于通过提供必要数据满足大量消费者的一次性需求的重要性。这类情况中，就应该将内容过滤器放在接收系统中。

如果你已经读过介绍内容丰富器的内容，那么观察内容过滤器的示例时可能会提不起精神。使用基于 Actor 的过滤器的方式与使用内容丰富器的方式非常相似，但两者的目的是相反的。尽管二者的目的不同，但内容过滤器的示例程序中也含有吸引人的东西。因为内容丰富器的示例程序处理的是向外发送的消息，所以我们使用内容过滤器示例程序处理收到的消息，并使用不同的方式将消息分派给最终的消息消费者。

```
package co.vaughnvernon.reactiveenterprise.contentfilter

import akka.actor._
import co.vaughnvernon.reactiveenterprise.CompletableApp

case class FilteredMessage(
  light: String,
  and: String,
  fluffy: String,
  message: String) {
  override def toString = {
    s"FilteredMessage(" + light + " " + and + " "
      + fluffy + " " + message + ")"
  }
}

case class UnfilteredPayload(largePayload: String)

object ContentFilter extends CompletableApp(3) {
  val messageExchangeDispatcher =
    system.actorOf(
      Props[MessageExchangeDispatcher],
      "messageExchangeDispatcher")

  messageExchangeDispatcher !
    UnfilteredPayload(
      "A very large message with complex
structure...")

  awaitCompletion
  println("RequestReply: is completed.")
}
```

```

}

class MessageExchangeDispatcher extends Actor {
  val messageContentFilter =
    context.actorOf(
      Props[MessageContentFilter],
      "messageContentFilter")

  def receive = {
    case message: UnfilteredPayload =>
      println("MessageExchangeDispatcher: "
        + "received unfiltered message: "
        + message.largePayload)
      messageContentFilter ! message
      ContentFilter.completedStep
    case message: FilteredMessage =>
      println("MessageExchangeDispatcher: dispatching: "
        + message)
      ContentFilter.completedStep
    case _ =>
      println("MessageExchangeDispatcher: unexpected")
  }
}

class MessageContentFilter extends Actor {
  def receive = {
    case message: UnfilteredPayload =>
      println("MessageContentFilter: "
        + "received unfiltered message: "
        + message.largePayload)
      // 进行过滤...
      sender ! FilteredMessage(
        "this", "feels", "so", "right")
      ContentFilter.completedStep
    case _ =>
      println("MessageContentFilter: unexpected")
  }
}

```

运行这个应用程序可以得到下面的结果：

```

MessageExchangeDispatcher: received unfiltered message:
A very large message with complex structure...
MessageContentFilter: received unfiltered message: A very
large message with complex structure...
MessageExchangeDispatcher: dispatching: FilteredMessage(

```

```
this feels so right)
RequestReply: is completed.
```

此处应考虑对象能力模型（Object-Capability model）的安全规则 [OCM]。其中的第二条规则提出了最强的安全模式：亲子关系。当一个父 Actor 对象创建一个子 Actor 对象时，父 Actor 对象就拥有了指向该子 Actor 对象的唯一引用。这确保了父 Actor 对象能够将子 Actor 对象任何被传送的消息隔离开，从而确保子 Actor 对象的安全性。除非通过设计使子 Actor 对象创建自己的子 Actor 对象，或者接收其他 Actor 对象发送的消息或向其他 Actor 对象发送消息（这些情况会受到其他对象能力模型规则的限制），否则其他 Actor 对象无法向该子 Actor 对象发送消息。

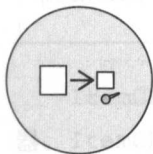
例如，内容丰富器的示例程序也符合对象能力模型，但组合使用了第 3 条和第 4 条规则。AccountingEnricherDispatcher 对象是通过 AccountingSystemDispatcher 对象封装的（符合第 3 条规则），而 ScheduledDoctorVisit 对象是由 AccountingEnricherDispatcher 对象引入的（符合第 4 条规则）。然而，内容过滤器示例程序使用了最强的安全保障，MessageContentFilter 对象不会转换 / 处理通过查找操作找到它的 Actor 对象发送的消息。

选择向 Actor 对象发送消息的方式的注意事项

在使用 Actor 模型时，某个 Actor 对象能够通过名称找到另一个 Actor 对象并向它发送消息，就是一种潜在的最危险情况。除非执行查找操作的是该 Actor 对象的父对象，否则这种方式会破坏对象能力模型的作用。

为什么父 Actor 对象不通过子 Actor 对象的 ActorRef 引用与子 Actor 对象进行交互，而使用查找操作呢？其原因可能是父 Actor 对象存储了过多子 Actor 对象的 ActorRef 引用，而且通过适当的查找操作能够节省系统内存，还能够降低仅通过引用管理子 Actor 对象的复杂程度。

存放证



当需要将一条组合消息分割成多个较小部分，并且还需要根据需求提供访问

这些部分的操作时，可使用存放证。

可以将存放证视为用于存储和访问被检查项目的唯一标识符。可以将指定被检查项目的标识符（即存放证），作为被处理消息的组成部分传输。处理过程中的每个步骤都可以根据需要使用存放证检索所有或部分组合消息内容。

可以将存放证模式与内容丰富器一起使用，以便重组全部消息成分、仅获取消息的部分内容，或执行这两者之间的操作。然而，此处的要点是基于消息的处理过程中的任何步骤都能够根据需要，检索原始消息内容中的一个或多个部分。实际上，本例没有使用存放证丰富消息的内容。该程序使用存放证获取原始消息的特定部分，这些部分都是指定处理步骤必须使用的部分，如图 8.3 所示。

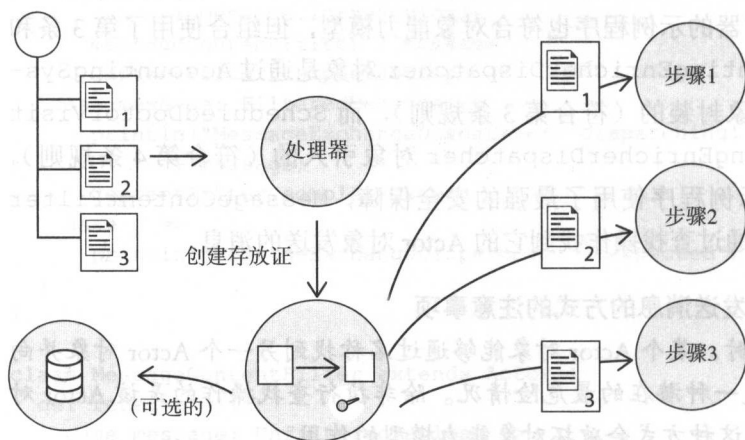


图 8.3 使用存放证可以获取原始消息的特定部分，但这些部分都是指定处理步骤必须使用的部分。

下面是存放证示例程序，其中包含了上述消息类型：

```
package co.vaughnvernon.reactiveenterprise.claimcheck

import akka.actor._
import co.vaughnvernon.reactiveenterprise._
import java.util.UUID

case class Part(name: String)

case class CompositeMessage(
  id: String,
  part1: Part,
  part2: Part,
  part3: Part)
```

```

case class ProcessStep(
  id: String,
  claimCheck: ClaimCheck)

case class StepCompleted(
  id: String,
  claimCheck: ClaimCheck,
  stepName: String)

object ClaimCheckDriver extends CompletableApp(3) {
  val itemChecker = new ItemChecker()

  val step1 =
    system.actorOf(
      Props(classOf[Step1], itemChecker),
      "step1")
  val step2 =
    system.actorOf(
      Props(classOf[Step2], itemChecker),
      "step2")
  val step3 =
    system.actorOf(
      Props(classOf[Step3], itemChecker),
      "step3")

  val process =
    system.actorOf(
      Props(classOf[Process],
        (Vector(step1, step2, step3),
          itemChecker)),
      "process")

  process ! CompositeMessage(
    "ABC",
    Part("partA1"),
    Part("partB2"),
    Part("partC3"))

  awaitCompletion
  println("ClaimCheck: is completed.")
}

```

ItemChecker 变量是用于保存存放证代表的被检查项目的简单数据存储
器。ItemChecker 没有实现为 Actor 对象，但该变量可以存储 Actor 对象。而且
ItemChecker 变量还可以使用后台数据库，但在本例中它仅将数据保存在内存中。

该处理过程中的每个步骤都由一个 Actor 对象代表。起控制作用的 Process 也是一个 Actor 对象，而且所有步骤都是通过 Vector 序列赋予它的。

下面是 ItemChecker、ClaimCheck、CheckedItem 和 CheckedPart 对象的代码：

```
import scala.collection.mutable.Map

case class ClaimCheck() {
  val number = UUID.randomUUID().toString

  override def toString = {
    "ClaimCheck(" + number + ")"
  }
}

case class CheckedItem(
  claimCheck: ClaimCheck,
  businessId: String,
  parts: Map[String, Any])

case class CheckedPart(
  claimCheck: ClaimCheck,
  partName: String,
  part: Any)

class ItemChecker {
  val checkedItems = Map[ClaimCheck, CheckedItem]()

  def checkedItemFor(
    businessId: String,
    parts: Map[String, Any]) = {
    CheckedItem(ClaimCheck(), businessId, parts)
  }

  def checkItem(item: CheckedItem) = {
    checkedItems.update(item.claimCheck, item)
  }

  def claimItem(claimCheck: ClaimCheck): CheckedItem = {
    checkedItems(claimCheck)
  }

  def claimPart(
    claimCheck: ClaimCheck,
    partName: String): CheckedPart = {
```

```

val checkedItem = checkedItems(claimCheck)

CheckedPart(
  claimCheck,
  partName,
  checkedItem.parts(partName))
}

def removeItem(claimCheck: ClaimCheck) = {
  if (checkedItems.contains(claimCheck)) {
    checkedItems.remove(claimCheck)
  }
}

```

ItemChecker 变量存储了全部 CheckedItem 实例。客户端可以查询全部 CheckedItem 实例或较小的 CheckedPart 对象。这种特殊的 ItemChecker 组件知道 CheckedItem 实例包含了 CheckedPart 实例，因此，ItemChecker 组件提供了聚合导航和查询功能。在基础存储器的顶部构建 ItemChecker 组件，可以为 Process 对象和它包含的步骤提供特定领域的查询功能。

Process 对象处理完所有被赋予它的步骤后，它的 CheckedItem 对象就会从 ItemChecker 变量中移除。下面让我们看看 Process 对象的代码：

```

class Process(
  steps: Vector[ActorRef],
  itemChecker: ItemChecker)
  extends Actor {

  var stepIndex = 0

  def receive = {
    case message: CompositeMessage =>
      val parts =
        Map(
          message.part1.name -> message.part1,
          message.part2.name -> message.part2,
          message.part3.name -> message.part3)

    val checkedItem =
      itemChecker
        .checkedItemFor(message.id, parts)

    itemChecker.removeItem(checkedItem)
  }
}

```

```

    steps(stepIndex) !
    ProcessStep(
        message.id,
        checkedItem.claimCheck)

case message: StepCompleted =>
    stepIndex += 1

if (stepIndex < steps.size) {
    steps(stepIndex) !
    ProcessStep(
        message.id,
        message.claimCheck)
} else {
    itemChecker.removeItem(message.claimCheck)
}

ClaimCheckApp.completedStep

case message: Any =>
    println(s"Process: unexpected: $message")
}
}

```

Process 对象被赋予了含有多个 ActorRef 引用的 Vector 序列，每个 ActorRef 引用代表处理过程中的一个步骤。Process 对象还被赋予了 ItemChecker 对象的引用。Process 对象可以根据收到的 CompositeMessage 消息，使用 ItemChecker 对象存储 CheckedItem 对象。之后，通过发送 ProcessStep 消息，可以执行每个处理步骤。当每个步骤收到 ProcessStep 消息时，会通过 ItemChecker 对象获取该步骤需要的 CheckedPart 对象。一旦处理步骤完成了，该步骤就会向该步骤当前正在处理的 ProcessStep 消息的发送者，发送 StepCompleted 事件消息。

所有 ProcessStep 消息都是由 Actor 对象 Process 发送的，这意味着 Process 对象也会接收所有 StepCompleted 事件消息。当收到一条 StepCompleted 消息时，Process 对象就会分派下一个步骤，直到所有步骤处理完毕为止。

下面的 3 个 Actor 对象都在 Process 对象中代表独立的步骤：

```

class Step1(itemChecker: ItemChecker) extends Actor {
    def receive = {
        case processStep: ProcessStep =>
            val claimedPart =
                itemChecker.claimPart(

```

```

processStep.claimCheck,
"partA1")

println(s"Step1: processing $processStep\n"
with $claimedPart")

sender !
  StepCompleted(
    processStep.id,
    processStep.claimCheck,
    "step1")

case message: Any =>
  println(s"Step1: unexpected: $message")
}

class Step2(itemChecker: ItemChecker) extends Actor {
  def receive = {
    case processStep: ProcessStep =>
      val claimedPart =
        itemChecker
          .claimPart(
            processStep.claimCheck,
            "partB2")

      println(s"Step2: processing $processStep\n"
with $claimedPart")

sender !
  StepCompleted(
    processStep.id,
    processStep.claimCheck,
    "step2")

case message: Any =>
  println(s"Step2: unexpected: $message")
}

class Step3(itemChecker: ItemChecker) extends Actor {
  def receive = {
    case processStep: ProcessStep =>
      val claimedPart =
        itemChecker.claimPart(
          processStep.claimCheck,
          "partC3")

```

```

println(s"Step3: processing $processStep\n"
with $claimedPart")

sender !
  StepCompleted(
    processStep.id,
    processStep.claimCheck,
    "step3")

case message: Any =>
  println(s"Step3: unexpected: $message")
}
}

```

根据该示例程序的输出结果所示, 处理过程中的每个步骤都声明了原始消息中的不同组成部分。消息中的每个组成部分都与特定的处理步骤对应。

```

Step1: processing ProcessStep(ABC,ClaimCheck(7b5d2d71-fcc1-4e94-8801-1f70397e0834))
with ClaimedPart(CheckedPart(CheckedItem(7b5d2d71-fcc1-4e94-8801-1f70397e0834),Part(partA1)))
Step2: processing ProcessStep(ABC,ClaimCheck(7b5d2d71-fcc1-4e94-8801-1f70397e0834))
with ClaimedPart(CheckedPart(CheckedItem(7b5d2d71-fcc1-4e94-8801-1f70397e0834),Part(partB2)))
Step3: processing ProcessStep(ABC,ClaimCheck(7b5d2d71-fcc1-4e94-8801-1f70397e0834))
with ClaimedPart(CheckedPart(CheckedItem(7b5d2d71-fcc1-4e94-8801-1f70397e0834),Part(partC3)))
ClaimCheck: is completed.

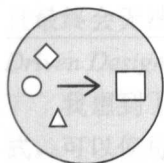
```

当然, 每个处理步骤都能够使用内容丰富器, 重组原始消息中某个较大的组成部分。这完全取决于特定步骤所需的数据量。在使用了内容丰富器的情况下, 内容丰富器会负责直接使用 `ItemChecker` 对象, 获取整个 `CheckedItem` 对象或任意数量的 `CheckedPart` 实例。

但为什么不将 `ItemChecker` 设计为 `Actor` 对象呢? 将 `ItemChecker` 设置为 `Actor` 对象可以取得很好的效果。但是, 本例选择了一种彻底的同步处理方式。这样就会使每个处理步骤都能够立刻对它收到的 `ProcessStep` 消息做出回应。这比将 `ItemChecker` 实现为 `Actor` 对象更好。当将 `ItemChecker` 实现为 `Actor` 对象时, 该对象会通过接收异步请求的方式, 接收 `CheckedItem` 或 `CheckedPart` 对象。毫无疑问, 将 `ItemChecker` 实现为远程 `Actor` 对象, 会使几乎所有处理步骤都

能够请求和接收 `CheckedItem` 或 `CheckedPart` 对象。几乎所有 `Actor` 对象都能够参与处理过程，是实现存放证的必要条件。

标准化器



当你编写的系统收到它不支持的类型的消息，并且需要将这类消息传送给其他支持这类消息的系统时，可使用标准化器。有时，某些外部系统或开发团队不支持你使用的消息数据格式，但你必须与他们进行整合。在这类情况中，你不得不将收到的自己系统不支持的消息，转换为自己系统能够支持的消息类型。*Implementing Domain-Driven Design* 一书中也通过防腐化层讨论过这方面的内容，如图 8.4 所示。

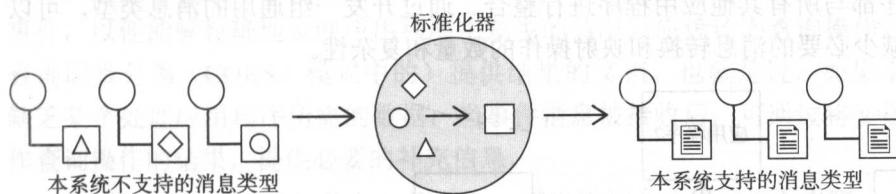


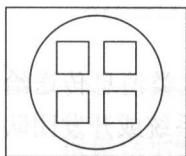
图 8.4 这个标准化器会将收到的本系统不支持的消息，转换为本系统支持的消息类型。

显然，当需要整合的系统很多并且仅有极少几个或没有系统愿意发送你编写的系统支持的类型的消息时，就需要使用大量的消息译码器。标准化器收到消息后，会使用消息路由器将本系统不支持的消息，分发给相应的消息译码器。如果收到的消息的格式或本系统的消息格式频繁改变，就会为标准化操作带来较大的挑战。而且，如果无法通过收到消息的内容轻松地识别出它们的格式，也会给标准化操作带来棘手的难题。因此，甚至在将消息转发给相应的译码器前，消息路由器本身就需要使用复杂的代码检测收到的消息的类型。简言之，这需要使用基于内容的路由器，但是还需要在其中添加简单的消息类型或格式表示器，才能使得该路由器处理比较高端的工作。

转换消息格式可能是一种需要外包的工作，遇到这类工作的数量很多并且需要进行不间断处理的情况时尤应如此。尽管使最终被处理的消息符合你使用的标

准格式这一点非常重要，但是对收到的消息进行大规模的检测和转换需要进行艰苦的工作和付出海量的精力。因为路由和转换代码几乎注定不会含有核心的业务模型，所以你可能不想将开发团队的宝贵精力投入到这项艰苦的无底洞般的工作中。你的开发团队解决了前几个路由器和译码器难题后，通过形成易于理解的处理方式和外包路由与转换工作，可以为开发团队带来很大的好处。

规范化消息模型



当需要将多个应用程序整合到一起，并且需要降低应用程序对其他应用程序所用数据类型的依赖性时，通常可使用规范化数据模型。请思考图 8.5 所示的 6 个应用程序。选择使用这种模式的一个主要原因是，这 6 个应用程序中的每一个应用程序都与所有其他应用程序进行整合。通过开发一组通用的消息类型，可以大幅度减少必要的消息转换和映射操作的数量和复杂性。

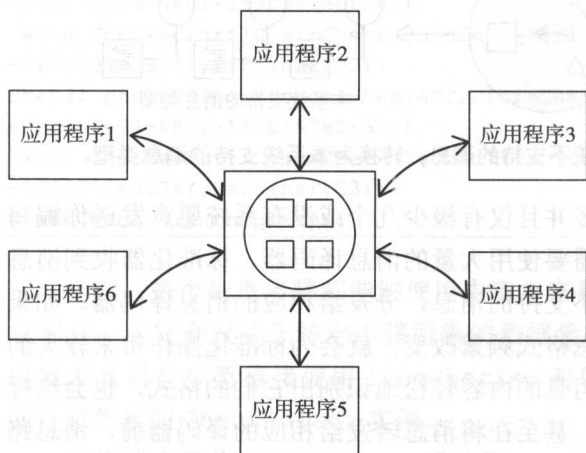


图 8.5 当需要将多个应用程序整合到一起时，可使用规范化消息模型。这 6 个应用程序使用了一组经过挑选的消息。

在整合如图 8.5 所示的应用程序时，如果每个应用程序都使用与其他应用程序不同的模型，就需要使用 6×5 个译码器（接收和发送消息使用不同的译码器）。然而，如果每个应用程序都使用与其他应用程序不同的模型，在使用规范化消息

模型的情况中,仅需要使用 12 个译码器。

但实际上,使用规范化数据模型支持上面介绍的跨应用程序模型,通常会徒劳无功。要使由各个应用程序(图 8.5 所示的 6 个应用程序)代表的各方,都同意使用一个较小的通用模型几乎是不可能的。因此,规范化数据模型通常最终成为各方会用到的所有对象和属性的超集。该模型从一开始就会笨拙不堪,而且最终会无法为任何一方提供良好的服务 [Tilkov-CDM]。Implementing Domain-Driven Design 一书中详细介绍了这方面内容。

我想到了另一种处理方式,将它称为规范化消息模型。通过使用这种处理方式,可以使每个应用程序(有界的上下文)的开发团队,自行决定使用哪种命令消息处理操作、使用哪种事件消息表示发生的情况,以及将哪种文档消息用作查询结果。通过将这些内容编写为合理的公共语言,可以使每个团队都受益。因此,可以将图 8.5 所示的模型视为由各个应用程序(有界上下文)开发团队承认的混合物。

这是一种自然得多的看待这种信息交换模型的方式,因为这可以将规范化数据模型大幅度简化为,各个应用程序都需要使用的命令消息、事件消息和文档消息。应设计所有应用程序都能够执行的命令。应设计具有适当名称和足够数据的事件,以便能够精确地反映应用程序中发生的情况。应设计为查询操作(如命令查询职责分离(CQRS)模式中的)提供结果的文档。也就是说,如果事件消息缺乏某个处理应用程序所需的数据,当事件消息被接收后,可通过将文档消息用作查询操作的结果,提供必要的补充信息。

Implementing Domain-Driven Design 一书也详细介绍了这方面内容。

Actor 系统需要标准

在使用 Actor 模型时,必须根据其他 Actor 对象的消息创建 Actor 对象。因此,需要仔细考虑交换消息的方式,和一个或各种应用程序中的哪些 Actor 对象需要依赖哪些应用程序的消息。因而,使用这种方式可以彻底避免消息转换处理过程,因为这种方式使 Actor 对象仅依赖外部的共享消息类型。

消息总线就是这样的方式。简言之,可将规范化消息模型中的所有通用消息都放入一个 Scala 源文件中,从而能够轻松将这些类型的消息部署到依赖它们的系统中。例如,下面是示例程序 Trading Bus 中使用的命令消息和事件消息,而且它们都被放在一个公用源文件中:

```
// 命令消息
case class ExecuteBuyOrder(portfolioId: String,
```

```

    symbol: Symbol, quantity: Int, price: Money)
case class ExecuteSellOrder(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
...
// 事件消息
case class BuyOrderExecuted(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
case class SellOrderExecuted(portfolioId: String,
    symbol: Symbol, quantity: Int, price: Money)
...

```

因为这些类型的消息必定会被至少一个 Actor 对象发送，并被至少一个 Actor 对象接收，所以将它们都放在一个公用的源文件中，能够更轻松地为依赖这些消息的 Actor 对象提供支持。请参阅第 4 章，详细了解这种方式的优缺点。

小结

本章详细介绍了在设计和整合应用程序时，使用的各种消息转换方式。所有这些方式都是对消息译码器的扩展，根据特定的应用程序需求提供专门的转换功能：封装、丰富化、过滤、跟踪、标准化和通用化。

第 9 章

消息端点

第 4 章介绍过消息端点就是 Actor 模型中的 Actor 对象。本章详细介绍各种端点。

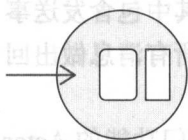
- **消息传输网关**：Akka 系统和其中的 Actor 对象会自然而然地形成消息传输网关。需要用于与消息传输系统（其中的 Actor 对象会相互发送消息）进行交互的代码非常简单。
- **消息传输映射**：使用消息传输映射，可以将一个或多个领域对象的组成部分与消息对应起来。
- **事务型客户端 / Actor 对象**：当通过 Akka 框架中的 Actor 对象使用这种模式时，对于客户端 / 发送者 Actor 对象和接收者 Actor 对象来说，这种模式是事务型的。
- **轮询消费者**：在该模式中，消费者会从指定的资源通过请求获取信息，而该资源能够在提供该信息前阻塞消费者。这与 Actor 模型的处理方式不同。使用请求—回复模式可以大致模拟轮询处理方式。使用 Akka 框架中的 Actor 对象，可以通过阻塞较少或无阻塞模式，谨慎地轮询硬件资源。
- **由事件驱动的消费者**：这种模式不是指发送事件消息，但其中包含发送事件消息的内容。由事件驱动的消费者是一种会对它收到的所有消息做出回应的 Actor 对象。
- **具有竞争性的消费者**：具有竞争性的消费者是一组提供专门功能的 Actor 对象，这些 Actor 对象会同时对多条消息做出回应。根据轮询消费者和消息调度器的实现方式，可以自然而然地创建具有竞争性的消费者。
- **消息调度器**：消息调度器与基于内容的路由器很相似。消息调度器会查看消息的内容（如消息类型），然后将消息分发给与消息类型相匹配的 Actor 对象。消息调度器与基于内容的路由器之间的差异，体现在对工作负荷的关注度上，消息调度器仅会将消息分发给能够对消息迅速做出回应的 Actor 对象。
- **选择性消费者**：选择性消费者是一种消息过滤器。如果某个 Actor 对象能

够接收各种类型的消息，但是只能处理其中的一部分，该 Actor 对象可以抛弃它无法处理的那些 Actor 对象。

- **持久订阅者**：当需要确保接收者 Actor 对象在不主动监听消息的情况下，不会错过已发送的消息时，可使用持久订阅者。
- **幂等接收者**：当某个 Actor 对象可能会多次接收同一条消息，而且每次处理这条消息都会出问题时，可以将该 Actor 对象设计为幂等接收者。因为 Actor 模型中的标准消息传输协定是至多发送一次，所以好像永远不会出现重复发送某一条消息的问题。然而，当混入 `AtLeastOnceDelivery` 特征时就会出现这类问题。
- **服务激活剂**：当内部服务需要使用一个或多个外部资源时，可使用服务激活剂。在应用程序的外部边界接收消息，并向内部的服务或领域模型中的 Actor 对象分发消息的 Actor 对象都是服务激活剂。

通过组合上面介绍的各种模式，可以实现响应式设计目标。如介绍轮询消费者的内容所示，实际上在创建响应式程序时既会用到具有竞争性的消费者，也会用到轮询消费者。在编写响应式程序时，消息调度器是工作提供者，消息调度器关注的是工作负载和响应能力，并且仅会将工作分配给表明自己能够处理该工作的消费者。根据工作消费者所做工作的类型，还需要使用事务型 Actor 对象，以便与聚合模式相符。

消息传输网关



Akka 系统和其中的 Actor 对象会自然而然地形成消息传输网关。Akka 框架使在 Actor 对象之间发送消息变得简单。在大多数情况中，不必在 Akka 系统和其中的 Actor 对象的上方创建另一个用于简化消息传输操作的抽象层。不论 Actor 对象是向本地 Actor 对象发送消息，还是向远程 Actor 对象发送消息，该接口都非常简单易用。

```
riskAssessment ! AttachDocument("This is a HIGH risk...")
```

`riskAssessment` 引用的 Actor 对象是本地的还是远程的呢？Akka 框架提供的部分消息传输网关能够在本地和远程 Actor 对象之间传输消息，其中包含了

由 ActorRef 和 RemoteActorRef 引用提供的抽象。当一台 Java 虚拟机 (JVM) 中的 Actor 对象向本地或远程 Actor 对象发送消息时, 不必知道目标 Actor 对象的地址。因此, 在某些情况中, 你不必查明 riskAssessment 是本地 Actor 对象还是远程 Actor 对象, 通常不必关心这个问题。

使用额外的消息传输网关的原因之一是, 需要实现 Akka 框架没有直接提供的功能。需要使一个带有实体或聚合对象特点 [IDDD] 的 Actor 对象, 拥有临时行为, 从而使该 Actor 对象能够根据某些条件 (如使用模式), 通过动态方式加载或卸载, 就是这类情况的一个例子。在这种情况下, Actor 对象拥有唯一身份和与可变状态对应的潜在功能, 而且可能当前正位于内存中或者将它的状态保存在硬盘中。如果客户端向该 Actor 对象发送了一条消息, 而且该 Actor 对象已经位于内存中, 那么该 Actor 对象就能够直接接收该条消息。如果当客户端向该 Actor 对象发送消息时, 该 Actor 对象没有位于内存中, 那么系统就必须在使该 Actor 对象接收消息前, 通过从硬盘读取数据重组该 Actor 对象, 将其加载到内存中。

下面是一个用于展示这种临时行为的 Actor 对象的示例。该应用程序的基本思路是通过聚合方式实现领域模型。下面是 DomainModel 类和它的伴生对象:

```
object DomainModel {
  def apply(name: String): DomainModel = {
    new DomainModel(name)
  }
}

class DomainModel(name: String) {
  val aggregateTypeRegistry =
    scala.collection.mutable.Map[String, AggregateType]()
  val system = ActorSystem(name)

  def aggregateOf(typeName: String, id: String): AggregateRef = {
    if (aggregateTypeRegistry.contains(typeName)) {
      val aggregateType = aggregateTypeRegistry(typeName)
      aggregateType.cacheActor ! RegisterAggregateId(id)
      AggregateRef(id, aggregateType.cacheActor)
    } else {
      throw new IllegalStateException(
        "DomainModel type registry does not have a $typeName")
    }
  }

  def registerAggregateType(typeName: String): Unit = {
    if (!aggregateTypeRegistry.contains(typeName)) {
      val actorRef = system.actorOf(
```

```

    Props(new AggregateCache(typeName)), typeName)
    aggregateTypeRegistry(typeName) = AggregateType(actorRef)
  }
}

def shutdown() = {
  system.shutdown()
}

...
case class AggregateType(cacheActor: ActorRef)

```

DomainModel 是一种基本抽象,用于管理聚合类型缓存。实际上,在本例中,每个聚合类型都含有缓存。如本示例应用程序 (DomainModelDriver) 的前半部分所示。

```

object DomainModelDriver extends CompletableApp(1) {

  val orderType = "co.vaughnvernon.reactiveenterprise.domainmodel.
Order"

  val model = DomainModel("OrderProcessing")

  model.registerAggregateType(orderType)

  val order = model.aggregateOf(orderType, "123")

  ...
}

```

创建了 DomainModel 实例后, Order 类型通过 DomainModel 对象被注册为聚合类型。然后,该程序命令 DomainModel 对象根据新注册的 Order 类型,提供聚合 Actor 实例。之后,DomainModel 对象生成了一个新的 Order 实例 (Actor 对象引用),该引用被赋予实体标识符 123。下面就是这个简单的聚合 Actor 对象 Order,除了提供用于接收消息的 Actor 对象外,该对象没有执行任何有意义的操作。

```

case class InitializeOrder(amount: Double)
case class ProcessOrder()

class Order extends Actor {
  var amount: Double = _

  def receive = {

```

```

case init: InitializeOrder =>
  println(s"Initializing Order with $init")
  this.amount = init.amount
case processOrder: ProcessOrder =>
  println(s"Processing Order is $processOrder")
  DomainModelPrototype.completedStep
}
}

```

然而，应注意由 DomainModel 对象的 aggregateOf() 函数返回的 Actor 对象引用 Order，不是我们熟悉的 ActorRef 类型，而是特殊的 AggregateRef 类型。这种 AggregateRef 引用的用法与 ActorRef 引用的用法非常相似，它是消息传输网关抽象中的关键，用于实现临时的聚合 Actor 对象。

让我们再看看应用程序 DomainModelDriver，一旦 DomainModel 对象提供了 Order 对象，就可以通过这个 AggregateRef 引用发送 Order 消息。

```

object DomainModelDriver extends CompletableApp(1) {

  val orderType = "co.vaughnvernon.reactiveenterprise.domainmodel.
Order"
  val model = DomainModel("OrderProcessing")

  model.registerAggregateType(orderType)

  val order = model.aggregateOf(orderType, "123")
  order ! InitializeOrder(249.95)
  order ! ProcessOrder()

  awaitCompletion
  model.shutdown()

  println("DomainModelDriver: is completed.")
}

```

然而，无法像使用 ActorRef 引用那样，直接将这些消息发送给 Order 对象。这些消息必须先通过 AggregateRef 引用抽象出来的消息传递网关。

```

case class AggregateRef(id: String, cache: ActorRef) {
  def tell(message: Any)(implicit sender: ActorRef = null): Unit = {
    cache ! CacheMessage(id, message, sender)
  }

  def !(message: Any)(implicit sender: ActorRef = null): Unit = {

```



```

    cache ! CacheMessage(id, message, sender)
  }
}

```

所有聚合 Actor 对象都是由特殊的缓存 Actor 对象 (AggregateCache 实例) 管理的。当 DomainModel 对象创建新的聚合 Actor 对象时, 会确保该聚合 Actor 对象由适当类型的 AggregateCache 实例管理。因此, 所有发送给聚合 Actor 对象的消息 (如 Order), 都会先被发送给起管理作用的缓存。

```

case class CacheMessage(
  id: String,
  actualMessage: Any,
  sender: ActorRef)

case class RegisterAggregateId(id: String)

class AggregateCache(typeName: String) extends Actor {
  val aggregateClass: Class[Actor] =
    Class
      .forName(typeName)
      .asInstanceOf[Class[Actor]]
  val aggregateIds =
    scala.collection.mutable.Set[String]()

  def receive = {
    case message: CacheMessage => {
      val aId = message.id
      val aggregate = context.child(aId).getOrElse {
        if (!aggregateIds.contains(aId)) {
          throw new IllegalStateException(
            s"No aggregate of type $typeName and id $aId")
        } else {
          context.actorOf(Props(aggregateClass), aId)
        }
      }
      // 重组聚合状态
      // 处理已存在的情况
    }
    aggregate.tell(message.actualMessage, message.sender)

    case register: RegisterAggregateId => {
      this.aggregateIds.add(register.id)
    }
  }
}

```

AggregateCache 对象负责处理两条消息：CacheMessage 和 RegisterAggregateId。当 DomainModel 对象创建了新的聚合 Actor 对象时，RegisterAggregateId 消息会先被处理。

```
class DomainModel(name: String) {
    ...
    def aggregateOf(
        typeName: String, id: String):
        AggregateRef = {
            if (aggregateTypeRegistry.contains(typeName)) {
                val aggregateType = aggregateTypeRegistry(typeName)
                aggregateType.cacheActor ! RegisterAggregateId(id)
                AggregateRef(id, aggregateType.cacheActor)
            } else {
                throw new IllegalStateException(
                    s"DomainModel registry doesn't have $typeName")
            }
        }
    ...
}
```

当 AggregateCache 对象收到 RegisterAggregateId 消息时，会存储新分配的唯一标识符。在本例中，所有唯一聚合标识符都存储在由 AggregateCache 对象管理的内存字段 Set 中。将所有这些标识符或其中的一部分存储到硬盘中，仅将 Set 字段中经常使用的标识符放在内存中，可以获得更好的效果。最终还是需要将所有这些标识符存储在硬盘中，以防某个 AggregateCache 对象崩溃。在这种处理方式中，永远都应跟踪聚合 Actor 对象，而且不要因 AggregateCache 对象认为某个聚合 Actor 对象不存在，而错误地重复创建该对象。

当 AggregateCache 对象收到 CacheMessage 消息时，会执行更为有趣的操作。CacheMessage 消息会通过 AggregateRef 引用被发送给 AggregateCache 对象。发送给聚合 Actor 对象的原始消息（如 InitializeOrder 和 ProcessMessage），是由 CacheMessage 消息封装的。AggregateCache 对象成功找到已缓存的聚合 Actor 对象后，第一次执行该操作时会创建该聚合 Actor 对象，以后会通过从硬盘读取数据重组该聚合 Actor 对象并缓存它，原始消息最终会被分发给该聚合 Actor 对象。

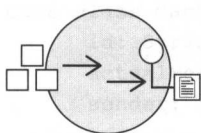
```
aggregate.tell(message.actualMessage, message.sender)
```

在这种处理方式中，接收消息的聚合 Actor 对象会像直接从消息的原始发送者接收消息一样接收消息，而不是通过 AggregateRef 引用从 AggregateCache

对象那里接收消息。因此，这个消息传输网关的目的不是简化已经简单的 Akka 消息发送语义。更确切地说，该消息传输网关用于实现一种 Akka 核心功能中没有的临时 Actor 对象缓存。

请参阅第 5 章，详细了解确保送达机制，以确保 Actor 对象（如聚合 Actor 对象）能够收到发送给它的所有消息。

消息传输映射



使用消息传输映射可以将一个或多个领域对象的组成部分（如聚合对象），与消息对应起来。根据你使用 Akka 框架的方式，可能你使用的聚合对象本身就是 Actor 对象，而能够将它们的部分状态，与事件消息、命令消息和文档消息对应起来。下面的例子展示了将事件消息与聚合 Actor 对象的状态对应起来，并发送给 TradingBus 对象的方式：

```
...
val event =
  SellOrderExecuted(
    portfolioId,
    symbol,
    quantity,
    price)

tradingBus ! TradingNotification(
  "SellOrderExecuted", event)
```

这样就将聚合 Actor 对象 Order 变成了一个消息传输映射。Actor 对象 Order 的部分状态与事件消息 SellOrderExecuted 对应了起来。这是一种简单直观的映射关系，而且实际上这是大多数情况中应该采用的方式。事件消息不应含有过多的字段/属性，但是应该含有表明 Order 对象所发生情况的所有必要数据。也应该使用这种方式处理命令消息，命令消息只应含有必要的用于描述命令的数据。

然而，当你考虑映射的类型时，有时需要考虑复杂的映射情况。例如，你想将聚合对象中的某些组成部分与较大的文档消息对应起来。虽然这是可能出现的

情况，但是应该尽量避免出现该情况。

为了回应命令查询职责分离（CQRS）查询操作 [IDDD]，可能需要使用被映射的文档消息创建得相当大的有效数据负荷。当出现这种情况时，就必须使用更为复杂的映射工具。令人遗憾的是，在 JVM 上运行的大多数映射工具，都会支持 JavaBeans 规范。这意味着你编写的基于 Scala 语言的消息对象需要提供公共读取器和写入器，否则就需要使映射工具支持字段级对应关系，当将消息声明为样本类时尤须如此。

解决这个问题的一种方式，是为文档消息设计一个基于 String 类型的字段（如 `messageBody`），这样就可以将这个字段设置为 JavaScript 对象表示法（JSON）格式或可扩展标记语言（XML）格式，进而使该字段能够被消费者端适当的解析器读取。*Implementing Domain-Driven Design* 一书详细介绍了创建与 Google GSON 解析器的各种系统平台兼容的消息的方式，该方式使用了字段级内省和对应关系。字段级访问操作支持任何类型的 Java/Scala 对象，包括那些不支持 JavaBeans 规范的对象。实际上，甚至可以将消息传输映射视为序列化器。

下面的源代码中既含有基于 Java 的 `AbstractSerializer` 的对象，也含有可用于将 Scala 样本类或任何类型的 Scala 对象与 JSON 格式数据对应起来的 `MessageSerializer` 对象。它们都使用了 Google GSON 解析器。

```
public abstract class AbstractSerializer {

    protected class CustomAdapter {
        private Object adapter;
        private Type type;

        CustomAdapter(Type aType, Object anAdapter) {
            this.type = aType;
            this.adapter = anAdapter;
        }

        protected Object adapter() {
            return this.adapter;
        }

        protected Type type() {
            return this.type;
        }
    }

    private Gson gson;
```

```

protected AbstractSerializer(boolean isCompact) {
    this(false, isCompact);
}

protected AbstractSerializer(
    boolean isPretty,
    boolean isCompact) {
    super();

    if (isPretty && isCompact) {
        this.buildForPrettyCompact();
    } else if (isCompact) {
        this.buildForCompact();
    } else {
        this.build();
    }
}

protected abstract Collection<CustomAdapter>
    customAdapters();

protected Gson gson() {
    return this.gson;
}

private void build() {
    this.gson =
        this
            .builderWithAdapters()
            .serializeNulls()
            .create();
}

private void buildForCompact() {
    this.gson = this.builderWithAdapters().create();
}

private void buildForPrettyCompact() {
    this.gson =
        this
            .builderWithAdapters()
            .setPrettyPrinting()
            .create();
}

private class DateSerializer
    implements JsonSerializer<Date> {

```

```

public JsonElement serialize(
    Date source,
    Type typeOfSource,
    JsonSerializerContext context) {
    return new JsonPrimitive(
        Long.toString(source.getTime()));
}

private class DateDeserializer
    implements JsonSerializer<Date> {
    public Date deserialize(
        JsonElement json,
        Type typeOfTarget,
        JsonSerializerContext context)
        throws JsonParseException {
        String nr = json.getAsJsonPrimitive().getAsString();
        long time = Long.parseLong(nr);
        return new Date(time);
    }
}

private GsonBuilder builderWithAdapters() {
    GsonBuilder builder = new GsonBuilder();

    builder.registerTypeAdapter(
        Date.class,
        new DateSerializer());
    builder.registerTypeAdapter(
        Date.class,
        new DateDeserializer());

    for (CustomAdapter adapter : this.customAdapters()) {
        builder
            .registerTypeAdapter(
                adapter.type(),
                adapter.adapter());
    }

    return builder;
}

public class MessageSerializer extends AbstractSerializer {

    public interface AggregateRefProvider {
        public AggregateRef fromTypeNameWithId(

```

```

        String typeName,
        String id);
    }

    private static MessageSerializer serializer;

    public static MessageSerializer instance() {
        if (serializer == null) {
            throw new IllegalStateException(
                "Must first use newInstance(...) before using"
                + "instance().");
        }
        return serializer;
    }

    public static synchronized MessageSerializer newInstance(
        AggregateRefProvider anAggregateRefProvider) {
        if (MessageSerializer.serializer == null) {
            MessageSerializer.serializer =
                new MessageSerializer(anAggregateRefProvider);
        }
        return MessageSerializer.serializer;
    }

    private AggregateRefProvider aggregateRefProvider;

    public String serialize(Object aMessage) {
        String serialization = this.gson().toJson(aMessage);

        return serialization;
    }

    public <T extends Object> T deserialize(
        String aSerialization, final Class<T> aType) {
        T message = this.gson().fromJson(aSerialization, aType);

        return message;
    }

    public <T extends Object> T deserializeOriginalMessage(
        String aSerialization, final Class<T> aType) {
        T message = this.gson().fromJson(aSerialization, aType);

        return message;
    }

```



```
@SuppressWarnings("unchecked")
```

```
public Class<Message> messageClassFrom(String aMessageType) {
    Class<Message> messageClass = null;
```

```
try {
```

```
    messageClass =
```

```
        (Class<Message>) Class.forName(aMessageType);
```

```
} catch (Exception e) {
```

```
    throw new IllegalArgumentException(
```

```
        "Cannot get class for message of type: "
        + aMessageType, e);
```

```
}
```

```
return messageClass;
```

```
}
```

```
@Override
```

```
protected Collection<CustomAdapter> customAdapters() {
```

```
    List<CustomAdapter> customAdapters =
        new ArrayList<CustomAdapter>();
```

```
customAdapters.add(
```

```
    new CustomAdapter(
```

```
        AggregateRef.class,
```

```
        new AggregateRefSerializer());
```

```
customAdapters.add(
```

```
    new CustomAdapter(
```

```
        AggregateRef.class,
```

```
        new AggregateRefDeserializer());
```

```
return customAdapters;
```

```
}
```

```
private MessageSerializer(boolean isCompact) {
```

```
    this(false, isCompact);
```

```
}
```

```
private MessageSerializer(
```

```
    boolean isPretty,
```

```
    boolean isCompact) {
```

```
    super(isPretty, isCompact);
```

```
}
```

```
private MessageSerializer(
```

```
    AggregateRefProvider anAggregateRefProvider) {
```

```

        this(false, false);

        this.aggregateRefProvider = anAggregateRefProvider;
    }

    private class AggregateRefSerializer
        implements JsonSerializer<AggregateRef> {

        public JsonElement serialize(
            AggregateRef source,
            Type typeOfSource,
            JsonSerializationContext context) {
            String cacheType = source.cache().path().name();
            String aggregateId = source.id();
            return new JsonPrimitive(
                cacheType + " " + aggregateId);
        }
    }

    private class AggregateRefDeserializer
        implements JsonDeserializer<AggregateRef> {

        public AggregateRef deserialize(
            JsonElement json,
            Type typeOfTarget,
            JsonDeserializationContext context)
            throws JsonParseException {
            String refParts =
                json.getAsJsonPrimitive().getString();
            int separator = refParts.indexOf(' ');
            if (separator == -1) {
                throw new IllegalStateException(
                    "The AggregateRef parts are incorrectly"
                    formatted: " + refParts);
            }

            return aggregateRefProvider.fromTypeNameWithId(
                refParts.substring(0, separator),
                refParts.substring(separator+1));
        }
    }
}

```

MessageSerializer 对象拥有序列化消息或任何 Scala 对象和为消息或任何 Scala 对象解除序列化的功能，这些 Scala 对象包括 java.util.Date 和 AggregateRef 值。要详细了解 AggregateRef 引用，请参阅前面介绍消息传

输网关的内容。

这个序列化器还能够通过从基于 String 类型的 `messageBody` 字段获取 JSON 格式的有效数据，组装出加大的文档消息。

```
case class ReallyBigQueryResult(messageBody: String)
```

```
class OrderQueryService extends Actor {
```

```
  val serializer = MessageSerializer.instance()
```

```
  ...
```

```
  def receive = {
```

```
    ...
```

```
    case query: QueryMonthlyOrdersFor =>
```

```
      val queryResult = monthlyOrdersFor(query.customerId)
```

```
      val messageBody = serializer.serialize(queryResult)
```

```
      sender ! ReallyBigQueryResult(messageBody)
```

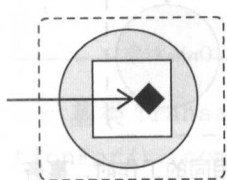
```
    ...
```

```
  }
```

```
}
```

这样 `QueryMonthlyOrdersFor` 消息的原始发送者，只需了解处理文档消息 `QueryResult` 中基于 JSON 的 `messageBody` 字段的方式。这只需要原始消息发送方的开发团队设计公共语言（如第 6 章所述），以及消息接收方的开发团队设计防腐化层（如第 4 章所述）。这样 `ReallyBigQueryResult` 消息和其他类似消息的消费者就能够读取有效数据了。

事务型客户端/ Actor对象



使用 Akka 框架中的 Actor 对象实现的事务型客户端模式，既含有客户端 Actor 对象的事务也含有接收者 Actor 对象的事务。从使用 Actor 模型的目的方面讲，这种模式被命名为事务型客户端 / Actor 对象。*Enterprise Integration Patterns* 一书介绍了在使用典型的消息传输中间件工具时，应用事务型客户端的 4 种方式。

- **发送—接收消息对**：启动一个事务，接收并处理第一条（接收的）消息，创建并发送第二条消息，然后提交事务。
- **消息分组**：启动一个事务，发送或接收分组中的所有消息，然后提交事务。
- **消息 / 数据库协调**：启动一个事务，接收一条消息，更新数据库，然后提交事务。或者，更新数据库并发送消息，以便向其他各方报告该更新情况，然后提交事务。
- **消息 / 流程协调**：使用一对请求—回复消息执行一项工作。启动一个事务，获取这项工作，发送请求消息，然后提交事务。或者，启动另一个事务，接收回复消息，完成或放弃这项工作，然后提交事务。

使用事务可以确保 Akka 消息的发送操作、消息回执和被持久化（持久化是指将数据保存到可永久保存的存储设备中）的 Actor 对象状态，不出上述 4 种语义范围。

请思考这些必要的事务处理步骤：(A) 为了确保任意一个 Actor 对象都能向另一个 Actor 对象发送消息，并且任意一个 Actor 对象都能接收另一个 Actor 对象发送的消息，就需要使用事务。(B) 为了确保接收消息的 Actor 对象能够收到消息并发送确认收到消息的回执，就需要使用事务。(C) 为了确保接收消息的 Actor 对象能够收到使其切换到新状态的消息，就需要使用事务。通常事务处理步骤 A 不会构成独立的事务，而事务处理步骤 B 和 C 会分别构成独立的事务，如图 9.1 所示。

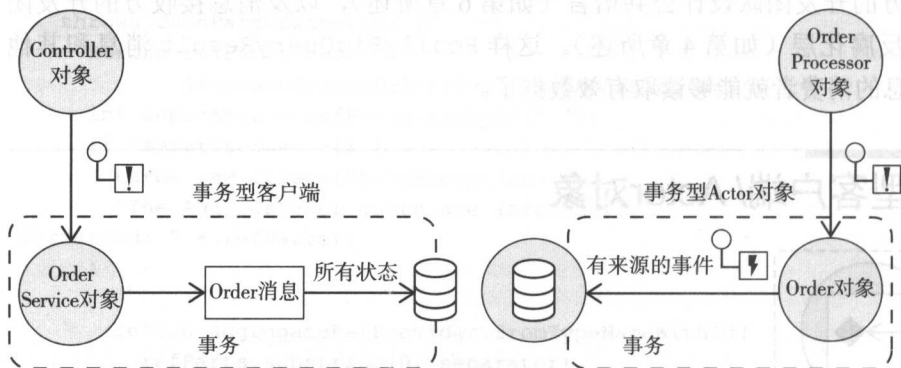


图 9.1 事务型客户端和事务型 Actor 对象是两种事务处理方式。在处理相同的工作时，事务型 Actor 对象最适合使用 Actor 模型。

可以通过向外部发送一条或多条事件消息实现事务处理步骤 C 中的状态事务（图 9.1 中被虚线框起来的部分）。在这种情况下，由 Actor 对象生成的事件消息可能还会被发送给其他 Actor 对象，从而形成前面介绍的第 3 种和第 4 种处理情况。

可通过几种方式根据切换的状态发送事务型消息和回执。要详细了解这些内

容, 请参阅第 5 章、第 9 章和第 10 章。而且, 因为事务处理步骤 A、B 和 C 都使用至少发送一次的消息发送机制, 所以你还需要了解设计幂等接收者的方式。

事务型客户端

你可能会选择从客户端的观点, 管理消息的发送、回执和事务状态切换。在使用服务层模式时 [Fowler EAA], 你会发现使用 Akka 框架默认的至多发送一次的消息发送机制 (而不是增强的至少发送一次机制, 如步骤 A 和 B 所用的), 同时使用事务处理方式 (如步骤 C) 管理持久化系统切换状态的操作, 会更为合适。在这类情况中, 你需要确定事务的起点和终点。

使用服务层模式 [Fowler EAA], 可以管理实现应用程序服务的 Actor 对象中的事务 [IDDD]。当收到消息时, 该 Actor 对象会启动一个数据库事务。这样基于 Actor 的应用程序服务就会直接与非 Actor 领域模型交互。然后, 该应用程序服务会提交或回滚该事务。

```
class OrderService extends TransactionalActor {
  def receive = {
    ...
    case processOrder: ProcessOrder =>
      startTransactionFor(processOrder.orderId)
      val order = findOrder(processOrder.orderId)
      try {
        order.process(processOrder) // 这不是一个 Actor 对象
        commitFor(processOrder.orderId)
      } catch {
        case _ =>
          rollbackFor(processOrder.orderId)
      }
    ...
  }
}
```

基类 TransactionalActor 提供了典型的事务型行为: startTransactionFor()、commitFor() 和 rollbackFor()。必须为这些函数提供具有唯一性的事务 ID, 因此可使用 Order 对象的唯一身份。当收到 ProcessOrder 消息时, 该事务就会启动。如果 Order 对象成功执行了 processOrder() 方法, 那么该事务就会被提交。否则, 该事务就会被回滚。将 orderId 字段用作该事务的 ID, 可以使每个 TransactionalActor 对象都能够同时管理多个事务。

如果这个 OrderService 对象能够按照预期方式运行, 事务型客户端就能够 在非 Actor 领域模型中引发任意数量的状态切换操作。但是应注意, 在一个事务

中改变的领域对象越多，出现并发冲突的可能性就越高，因为这会多次同时修改同一批领域对象。请参阅前面介绍聚合对象的内容，了解优化事务设计的最佳方式。

能够帮助处理这种情况的一个好习惯是，使用最终一致性（请参阅本章后面“最终一致性”小节，详细了解这方面的内容），通过在 Actor 模型中直接使用异步消息传输方式可以支持该处理方式。参与这类长期运行事务的所有应用程序服务，都会收到反映先前发生情况的事件消息，这些事件消息会告诉接收者必须执行哪些操作，以便支持最终一致性处理过程。

事务型客户端模式并不完全与 Actor 模型的概念相符，Actor 模型支持高等级的并发处理模式并且会将所有事务都视为 Actor 对象。通过事务型客户端使用 Actor 模型，很可能不如像通过 Actor 对象使用 Actor 模型那样顺手。但你会发现事务型客户端确实会争抢数据库资源。因此，在领域模型中将所有 Actor 对象都设计成 Actor 模型中的模范公民，可获得好得多的效果。这就推出了第二种事务处理模式，这种模式将领域模型中的所有 Actor 对象，都设计为独立管理自身事务的 Actor 对象。

事务型 Actor 对象

Alan Kay 是 Smalltalk 语言的发明者之一，也是一位面向对象技术的先驱。他说过：“Actor 模型保持的面向对象编程优点，比我原来设想的更多。”他提出，只有通过消息传输协定才能彻底在 Actor 对象中隔离状态管理操作，和使多个 Actor 对象协同工作。这间接指出了每个 Actor 对象周围的事务边界。

这不是说在持久性事务接收消息时，所有 Actor 对象都是事务型 Actor 对象。然而，可以将收到的每条消息，视为对指定消息刺激源的独立的、原子式的回应。当根据这些独立的、原子式回应管理持久性事务时，Actor 对象本身就是真正的事务型 Actor 对象。在这类情况中，Actor 对象会自然而然地执行聚合操作，下面详细介绍这些内容。

为了帮助处理这类情况，Akka 框架提供了名为 Akka Persistence 的持久性组件。要在项目中使用 Akka Persistence，只需将下列 JAR 文件添加到 akka/lib 目录中。

- akka-persistence_x.y.z.jar : Akka Persistence 组件。在我撰写本书时，该文件还处于试验阶段，当时的最新版本为 akka-persistence-experimental_2.10-2.3.2.jar。
- leveldb-x.y.jar : LevelDB 数据库功能。
- leveldb-api-x.y.jar : LevelDB 应用程序编程接口 (API)。
- leveldbjni-all-x.y.jar : LevelDB Java 本地接口 (JNI) 链接。

■ `protobuf-java-x.y.z.jar` : Java 版本的 Google ProtoBuf 库。

上面的文件名中都含有 `x.y.z`，它们代表版本号。尽管 LevelDB 是默认的存储格式，但并不是说产品级的消息存储器必须使用这种数据库。使用各种第三方软件也可以完成这项数据存储任务。然而，上面介绍的 JAR 文件能够帮助你更好地使用 Akka Persistence 组件。

持久性 Actor 对象

通过扩展 Akka 特征 `PersistentActor`，可以获得持久性 Actor 对象。这样做可以使 Actor 对象将它内部的事件持久化为领域事件流，这种模式称为事件溯源。事件溯源将每个已持久化的事件着重定义为反映改变情况（而不是整个 Actor 对象的状态）的记录。通过事件溯源模式，可以使由于各种原因（如接到监督者的命令或在平衡负荷时被分配到新的集群分片位置）停止运行的 Actor 对象，能够通过它最近持久化的事件流彻底进行重组。下面展示了持久性 Actor 对象的运行方式：

```
class Order(orderId: String) extends PersistentActor {

  override def persistenceId = orderId
  var open = false
  var lineItems = Vector[LineItem]()

  override def receiveCommand: Receive = {
    case cmd: StartOrder =>
      persist(OrderStarted(orderId, ...)) { event =>
        updateWith(event)
      }
    case cmd: AddOrderLineItem =>
      if (open) {
        val orderLineItemAdded =
          OrderLineItemAdded(orderId, ...)
        persist(orderLineItemAdded) { event =>
          updateWith(event)
        }
      }
    case cmd: PlaceOrder =>
      persist(OrderPlaced(orderId, ...)) { event =>
        updateWith(event)
      }
  }

  override def receiveRecover: Receive = {
```



```

    case event: OrderStarted =>
      updateWith(event)
    case event: OrderLineItemAdded =>
      updateWith(event)
    case event: OrderPlaced =>
      updateWith(event)
    case RecoveryCompleted =>
  }

  def updateWith(event: OrderStarted) = {
    open = true
  }

  def updateWith(event: OrderLineItemAdded) = {
    lineItems = lineItems :+ event.lineItem
  }

  def updateWith(event: OrderPlaced) = {
    open = false
  }
}

```

每个持久性 Actor 对象都必须使用具有唯一性的持久性标识符。本例将具有唯一性的持久性标识符 `orderId`，用作构造器参数。

`Order` 是一个持久性 Actor 对象，它通过它的 `receiveCommand` 处理程序代码块接收命令消息。收到 `StartOrder` 命令后，`Order` 对象会将一条 `OrderStarted` 事件消息持久化起来。一旦成功执行了 `persist()` 方法，`Order` 对象就会使用 `OrderStarted` 事件更新它的状态。此时，`Order` 对象会被标记为 `open`。之后 `Order` 对象接收的命令不是 `AddOrderLineItem` 就是 `PlaceOrder`。每次收到 `AddOrderLineItem` 命令消息后，如果 `Order` 对象仍旧被标记为 `open`，那么 `Order` 对象就会更新自己的状态，以便容纳已持久化的 `OrderLineItemAdded` 事件中包含的 `LineItem` 对象。最后，当 `Order` 对象收到 `PlaceOrder` 命令时，会持久化 `OrderPlaced` 事件，然后使用该事件关闭自己。

如果由于某种原因 Actor 对象 `Order` 被停止，然后又重启了，那么 `PersistentActor` 特征就会使 `Order` 对象能够通过 `Order` 对象的事件流重组自己。所有事件消息都是由 `receiveRecover` 处理程序代码块接收的，从而将 3 个 `updateWith()` 方法其中之一，应用于与 `Order` 对象的状态对应的事件。一旦 `Order` 对象完全恢复到事件流中最后一个事件消息代表的状态，`receiveRecover` 代码块就会收到标准的 `RecoveryCompleted` 事件。这个标准事件使 `Order` 对象能够在收到新的命令消息前，在完全恢复状态后这段时间中，执行任

何附加的特殊初始化操作。

令人鼓舞的是，实现持久化 Actor 对象非常简单。然而，多了解一点事件溯源（包括使用快照的方式），仍旧会使你受益不少。

使用事件溯源模式 在事件溯源模式中，对象或记录的状态是由对象或记录中曾经出现过的事件构成的。如图 9.2 所示，Actor 对象 Order 中出现过 4 个事件，而且该对象的状态是根据这些事件切换的。

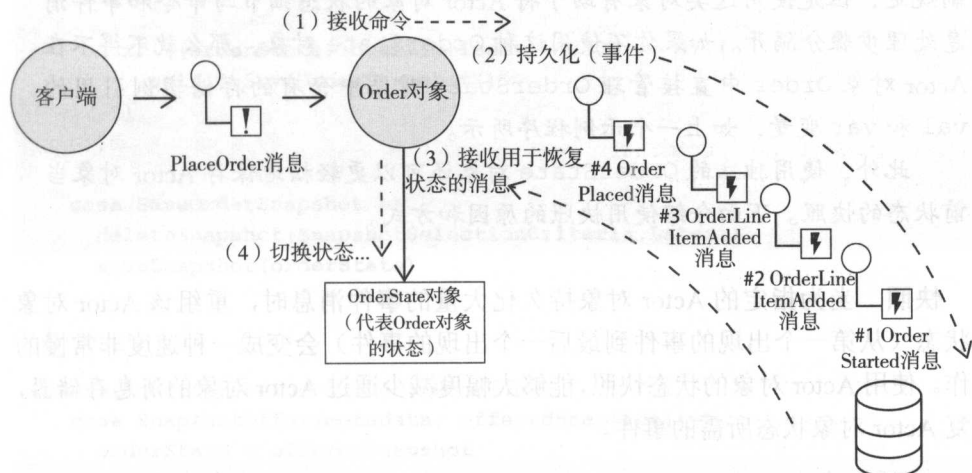


图 9.2 一个事务型 Actor 对象会接收命令消息并持久化它的新状态（步骤 1 和 2）。在恢复状态过程中，先前被持久化的领域事件被读取出来，被用于恢复状态（步骤 3 和 4）。

由 Order 对象持久化的所有事件消息，都是由相应的命令消息激发的，如图 9.2 所示。

1. Order 对象被创建并启动后，会收到 StartOrder 命令，该命令会激发 OrderStarted 事件。
2. 用户通过向 Order 对象发送 AddOrderLineItem 命令选择他要购买的商品，该命令会激发 OrderLineItemAdded 事件。
3. 用户通过向 Order 对象发送 AddOrderLineItem 命令选择他要购买的另一件商品，该命令会激发另一个 OrderLineItemAdded 事件。
4. 最后，用户通过向 Order 对象发送 PlaceOrder 命令，完成并提交订单，该命令会激发 OrderPlaced 事件。

持久化事件消息并非处理工作的全部。Actor 对象 Order 还必须对每个事件做出回应，这样它才能根据不同事件切换状态。每条事件消息必须以它们发生的次序被接收，从第 1 个事件到第 4 个事件，以此类推。

通过这种方式接收事件不仅能够使 Actor 对象切换状态，还能够使 Actor 对象的状态在 Actor 对象必须停止然后重启的情况下，能够通过已持久化的事件彻底重组。

注意

使 Actor 对象 Order 含有独立的代表状态的 OrderState 对象不是强制规定，但是使用这类对象有助于将 Actor 对象的状态细节与命令和事件消息处理步骤分隔开。如果你不使用这种 OrderState 对象，那么就不得不在 Actor 对象 Order 中直接管理 OrderState 对象中含有的存储实例引用的 val 和 var 变量，如上一个示例程序所示。

此外，使用独立的 OrderState 对象还可以更轻松地保存 Actor 对象当前状态的快照。下面介绍使用快照的原因和方式。

快照 当为指定的 Actor 对象持久化大量的事件消息时，重组该 Actor 对象的状态（从第一个出现的事件到最后一个出现的事件）会变成一种速度非常慢的操作。使用 Actor 对象的状态快照，能够大幅度减少通过 Actor 对象的消息存储器，恢复 Actor 对象状态所需的事件。

注意

使用事件溯源模式支持快照并非硬性规定。例如，如果某个 Order 对象拥有较少的订单来源，而且该 Order 对象持久化的事件消息总数大约为 200 个左右，那么就不必使用快照保存该 Order 对象的状态。另一方面，如果 Order 实例拥有数百乃至数千个订单来源，那么毫无疑问使用快照保存 Order 对象的状态，能够大幅度减少加载 Order 对象所需的时间。

世上不存在通用的指导原则。必须分析每个事件溯源 Actor 类型，以确定是否应使用快照。在得到肯定答案后，通常必须持久化快照。

在对持久性 Actor 对象使用快照时，必须指明在哪些情况下持久化快照。可以在处理指定的命令消息时持久化快照，也可以使持久性 Actor 对象通过向自己发送保存快照的消息持久化快照。

```
class Order(orderId: String) extends PersistentActor {
```

```
  override def persistenceId = orderId
  var orderState: OrderState(orderId)
```

```

override def receiveCommand: Receive = {
  ...
  case cmd: AddOrderLineItem =>
    if (orderState.open) {
      val orderLineItemAdded =
        OrderLineItemAdded(orderId, ...)
      persist(orderLineItemAdded) { event =>
        updateWith(event)
      }
      if ((orderState.totalLineItems % 250) == 0) {
        self ! SaveOrderSnapshot()
      }
    }

  case SaveOrderSnapshot =>
    deleteSnapshot(SnapshotSelectionCriteria.Latest)
    saveSnapshot(orderState)
  ...
}

override def receiveRecover: Receive = {
  case SnapshotOffer(metadata, offeredSnapshot) =>
    orderState = offeredSnapshot
  case event: OrderStarted =>
    updateWith(event)
  case event: OrderLineItemAdded =>
    updateWith(event)
  case event: OrderPlaced =>
    updateWith(event)
  case RecoveryCompleted =>
  ...
}

```

这个示例程序展示了管理快照的几个必要步骤。注意，本例使用了 Order-State 对象，从而使处理快照的工作变得更容易。

首先，Order 对象确定每收到第 250 个代表商品的 LineItem 对象（即第 250 个、第 500 个、第 750 个，以此类推）时，就需要保存一张快照。Order 对象会向自己发送一条 SaveOrderSnapshot 消息，以便命令自己保存快照。当收到 SaveOrderSnapshot 命令消息时，Order 对象会删除先前保存的快照（在存在先前保存的快照的情况下），然后保存一张新快照。

最后，在恢复 Order 对象的过程中，如果先前保存了该对象的快照，那么作为持久性 Actor 对象，Order 对象就能够恢复到该快照代表的时间点。之后，只有

比该快照更新的事件消息才能被发送给 `receiveRecover` 代码块，并被应用于 `orderState` 对象。

最终一致性

在基于 Actor 的领域模型中，还需要处理另一个重要方面。当切换基于 Actor 的聚合对象的状态时，另一个基于 Actor 的聚合对象可能与该切换操作有依赖关系。因为每个持久性 Actor 对象都自行管理自己的事务，实际上命令相互依赖的 Actor 对象同时更新它们的状态是不可能的。这事实上是一件好事，因为这有助于领域模型通过更好的事务型结果获得成功。

即便如此，所有依赖指定 Actor 对象的修改操作的 Actor 对象，都必须在指定的时限内更新。要做到这一点，就需要使用最终一致性。为什么会出现这种情况呢？因为已修改的 Actor 对象会争着处理特定的命令消息，并且会生成反映该 Actor 对象中已发生情况的事件消息（如 `OrderPlaced`）。这条事件消息会被发送给与该 Actor 对象有依赖关系的一个或多个 Actor 对象，然后这些 Actor 对象会根据这条事件消息切换自己的状态。如果它们之间的依赖关系不太强，那么仅发送必要的关键事件消息就足够了。然而，在管理长期运行的处理过程时，情况会变得更为复杂，很可能需要使用处理过程管理器。实际上，在通过 Actor 模型使用由领域驱动的设计模式时，很可能需要大量使用处理过程管理器，以便更精细地控制业务处理过程模型。

持久性视图

在我撰写本书时，Akka Persistence 插件支持了持久性视图。持久性视图是一种使持久性 Actor 对象的状态非规范化的方式，它能够使持久性 Actor 对象的状态更易于为应用程序逻辑观所用。然而，持久性视图不会永久存在，而且正要被使用 Akka Streams 的解决方案替代。当时 Akka Streams 解决方案还没有启动，而且它也没有确切的完成时间表。我面临的选择是，要么等待该解决方案被完成，要么先写完这本书。为了在 Akka Streams 解决方案完成后介绍它，我决定撰写博客文章，以作为本书内容的补充。

此处我仅会概要介绍持久性视图，通过参阅后面介绍持久订阅者的内容，你可以详细了解它。下面的代码展示了持久性视图处理方式：

```
class OrderView(orderId: String) extends PersistentView {

  override def persistenceId = orderId
  override def viewId = orderId + "--view"
  var viewState = OrderViewState()
```

```

override def receive: Receive = {
  case event: OrderStarted =>
    viewState = viewState.startedWith(event)
  case event: OrderLineItemAdded =>
    viewState = viewState.addTo(event)
    if ((viewState.totalLineItems % 250) == 0) {
      self ! SaveOrderViewSnapshot()
    }
  case event: OrderPlaced =>
    viewState = viewState.placedWith(event)
  case QueryOrderViewState =>
    sender ! viewState
  case SaveOrderSnapshot =>
    deleteSnapshot(SnapshotSelectionCriteria.Latest)
    saveSnapshot(viewState)
}

override def receiveRecover: Receive = {
  case SnapshotOffer(metadata, offeredSnapshot) =>
    viewState = offeredSnapshot
  case event: OrderStarted =>
    viewState = viewState.startedWith(event)
  case event: OrderLineItemAdded =>
    viewState = viewState.addTo(event)
  case event: OrderPlaced =>
    viewState = viewState.placedWith(event)
  case RecoveryCompleted =>
}

```

在持久性视图处理方式中，Actor 对象必须使用与相应持久性 Actor 对象相同的具有唯一性的持久性标识符，从存储设备中读取它的事件流。要将持久性视图标识符与持久性 Actor 对象标识符区分开，可在持久性 Actor 对象标识符后面加上 -view 后缀，以表示这是一个持久性视图的标识符。

receive 处理程序代码块会回应所有被相应的持久性 Actor 对象持久化的事件消息。你很可能需要保持非规范化的数据结构，如本例中的 OrderViewState 对象所示。像处理 Order 对象的示例一样，OrderView 对象也会在每收到 250 个 LineItem 对象后保存它的状态快照。此外，OrderView 对象还允许通过向它发送 QueryOrderViewState 消息，查询它的状态。而且它会向 QueryOrderViewState 消息的发送者，回复它不可变的 OrderViewState 实例。

在处理恢复过程时，持久性视图也会使用 receiveRecover 代码块。该代

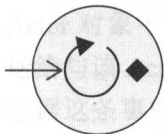
码块必须对 SnapshotOffer 对象和 3 个可能出现的事件之一做出回应，这 3 个事件包括：OrderStarted、OrderLineItemAdded 和 OrderPlaced。

你可能想知道持久性视图 Actor 对象使用最近持久化的事件，以怎样的频率刷新状态。通过下面的标准配置可以查明该频率，该配置表明该频率为每 5 秒一次：

```
akka.persistence.view.auto-update-interval = 5s
```

通过重写持久性视图 Actor 对象中的 autoUpdateInterval() 方法，可以自定义这个配置。

轮询消费者



在这种模式中，消费者通过轮询方式向指定资源请求获取信息。在资源能够提供该信息前，需要阻塞消费者。与此相反，在使用 Actor 模型时，无法使一个 Actor 对象以轮询方式向另一个 Actor 对象请求信息，因为 Actor 对象之间的协同操作不会被阻塞。一个 Actor 对象从另一个 Actor 对象获取信息的唯一方式是使用请求—回复模式。也就是说，需要使用请求—回复模式高效地模拟轮询消费者模式。此外，还需要使用 Actor 对象模拟不是由事件驱动的资源（如设备），以便使消费者能够以轮询方式通过这些资源获取信息。

在典型的过程化轮询环境中，工作消费者会从工作提供者那里请求获取工作。在工作提供者能够将被请求的工作分配给工作消费者前，工作消费者会被阻塞。Actor 模型中肯定不会出现这种情况。在使用 Actor 对象时，当工作消费者告诉工作提供者它需要获得工作时，工作消费者仍旧会在它本身的线程中继续运行，而且被发送给工作提供者的请求，过一段时间（这段时间可能长一点也可能短一点）才会被收到。不论接收和处理请求所需实际时间有多久，过程化轮询模式要求的在分配和回复被请求的工作时，使用的阻塞操作都无法被实现。

然而，仍旧可以使用请求—回复模式实现与轮询类似的效果，如图 9.3 所示。你需要做的是，将工作消费者设计为从工作提供者那里请求工作，然后获取该工作。

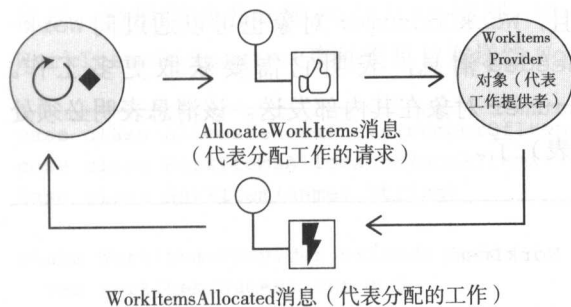


图 9.3 代表工作消费者的 Actor 对象，从代表工作提供者的 Actor 对象那里请求获取工作。工作提供者 Actor 对象通过为工作消费者 Actor 对象分配工作，对该请求做出了回复。

下面的代码展示了实现细节：

```

package co.vaghnvernon.reactiveenterprise.pollingconsumer

import scala.collection.immutable.List
import akka.actor._
import co.vaghnvernon.reactiveenterprise._

object PollingConsumerDriver extends CompletableApp(1) {
  val workItemsProvider =
    system.actorOf(
      Props[WorkItemsProvider],
      "workItemsProvider")
  val workConsumer =
    system.actorOf(
      Props(classOf[WorkConsumer],
        workItemsProvider),
      "workConsumer")

  workConsumer ! WorkNeeded()

  awaitCompletion
}
  
```

本示例应用程序 (PollingConsumerDriver) 创建了 Actor 对象 WorkItemsProvider 和 WorkConsumer (代表工作消费者)。然后该程序启动了工作处理过程，并通过向 WorkConsumer 对象发送 WorkNeeded 消息，命令 WorkConsumer 对象去获取工作。

WorkConsumer 类在本身的协定中声明了两种消息：WorkNeeded 和 WorkOnItem。客户端可以通过向 WorkConsumer 对象发送 WorkNeeded 消息，

启动该对象内部的处理过程。而且，WorkConsumer 对象也可以通过向 WorkItemsProvider 对象发送 WorkNeeded 消息，表明它需要获取更多工作。WorkOnItem 消息只能由 WorkConsumer 对象在其内部发送。该消息表明必须处理某项工作（由 WorkItem 对象代表）了。

```

case class WorkNeeded()
case class WorkOnItem(workItem: WorkItem)

class WorkConsumer(workItemsProvider: ActorRef)
  extends Actor {
    var totalItemsWorkedOn = 0

    def performWorkOn(workItem: WorkItem) = {
      totalItemsWorkedOn = totalItemsWorkedOn + 1
      if (totalItemsWorkedOn >= 15) {
        context.stop(self)
        PollingConsumerDriver.completeAll
      }
    }

    override def postStop() = {
      context.stop(workItemsProvider)
    }

    def receive = {
      case allocated: WorkItemsAllocated =>
        println("WorkItemsAllocated...")
        allocated.workItems map { workItem =>
          self ! WorkOnItem(workItem)
        }
        self ! WorkNeeded()
      case workNeeded: WorkNeeded =>
        println("WorkNeeded...")
        workItemsProvider ! AllocateWorkItems(5)
      case workOnItem: WorkOnItem =>
        println(s"Performed work on: ${workOnItem.workItem.name}")
        performWorkOn(workOnItem.workItem)
    }
  }

```

当 WorkConsumer 对象收到 WorkNeeded 消息后，会向在初始化时获得的 WorkItemsProvider 对象请求获取工作。为了做到这一点，WorkConsumer 对象会发送 AllocateWorkItems 消息。WorkItemsProvider 对象收到这条消

息后，会根据请求分配 WorkItem 实例。WorkItem 实例被分配后，就会通过 WorkItemsAllocated 事件消息被发送给 WorkConsumer 对象。

```

case class AllocateWorkItems(numberOfItems: Int)
case class WorkItemsAllocated(workItems: List[WorkItem])
case class WorkItem(name: String)

class WorkItemsProvider extends Actor {
  var workItemsNamed: Int = 0

  def allocateWorkItems(
    numberOfItems: Int): List[WorkItem] = {
    var allocatedWorkItems = List[WorkItem]()
    for (itemCount <- 1 to numberOfItems) {
      val nameIndex = workItemsNamed + itemCount
      allocatedWorkItems =
        allocatedWorkItems :+
          WorkItem("WorkItem" + nameIndex)
    }
    workItemsNamed = workItemsNamed + numberOfItems
    allocatedWorkItems
  }

  def receive = {
    case request: AllocateWorkItems =>
      sender !
        WorkItemsAllocated(
          allocateWorkItems(
            request.numberOfItems))
  }
}

```

让我们回过头再看 WorkConsumer 对象，当它收到 WorkItemsAllocated 消息时，会通过为每个 WorkItem 对象向自己发送这种消息，将这些工作划分为单个的 WorkOnItem 任务。然后，为了完成对 WorkItemsAllocated 消息的回应操作，WorkConsumer 对象会向自己发送一条 WorkNeeded 刷新消息。只有当所有 WorkOnItem 任务都被处理完后，WorkConsumer 对象才会接收这条 WorkNeeded 刷新消息。下面是这个模拟轮询处理过程的应用程序的输出结果：

```

WorkNeeded...
WorkItemsAllocated...
Performed work on: WorkItem1
Performed work on: WorkItem2

```

```

Performed work on: WorkItem3
Performed work on: WorkItem4
Performed work on: WorkItem5
WorkNeeded...
WorkItemsAllocated...
Performed work on: WorkItem6
Performed work on: WorkItem7
Performed work on: WorkItem8
Performed work on: WorkItem9
Performed work on: WorkItem10
WorkNeeded...
WorkItemsAllocated...
Performed work on: WorkItem11
Performed work on: WorkItem12
Performed work on: WorkItem13
Performed work on: WorkItem14
Performed work on: WorkItem15

```

当 15 个 WorkItem 任务都被处理完后, WorkConsumer 对象就会停止运行。最后, WorkConsumer 对象的 postStop() 函数中的 WorkItemsProvider 对象也会停止运行。

本示例程序仅创建了一个 WorkConsumer 实例。然而, 仅凭想象就可以知道, 应该根据多核计算机的核心数量, 为每个核心创建一个 WorkConsumer 对象。这样就可以使所有 Actor 对象 WorkConsumer 同时处理工作了。

这实际上引入了一种消息调度器模式。当需要获取工作时, 每个 Actor 对象 WorkConsumer 都必须通知 WorkItemsProvider 对象。这就需要使工作消费者知道工作提供者的地址。而且因为 WorkConsumer 对象必须向 WorkItemsProvider 对象发送 AllocateWorkItems 消息, 所以还需要发送更多消息 (如使用 Akka 标准 Actor 对象 BalancingDispatcher)。但轮询消费者模式节省了使用一种或多种 Akka 标准工具 (在后面介绍消息调度器时介绍) 时需要花费的检查系统开销, 而且也不要求你深入了解 Akka 框架。

资源轮询

前面介绍了使用 Actor 对象模拟轮询消费者模式的方式。但是, 你可能会遇到必须使用基于 Actor 的轮询消费者通过轮询非 Actor 资源获取信息的情况。这是特别难以处理的情况, 因为除非仔细小心地设计访问操作, 否则执行轮询操作的 Actor 对象所在的线程会在获取资源时被长时间阻塞, 甚至会导致出现系统不响应的情况。

下面的示例程序监听了一个特殊的设备（由 `EvenNumberDevice` 对象代表）。该设备用于提供偶数。

```
class EvenNumberDevice() {
    val random = new Random(99999)

    def nextEvenNumber(waitFor: Int): Option[Int] = {
        val timeout = new Timeout(waitFor)
        var nextEvenNumber: Option[Int] = None

        while (!timeout.isTimedOut && nextEvenNumber.isEmpty) {
            Thread.sleep(waitFor / 2)

            val number = random.nextInt(100000)

            if (number % 2 == 0) nextEvenNumber = Option(number)
        }

        nextEvenNumber
    }

    def nextEvenNumber(): Option[Int] = {
        nextEvenNumber(-1)
    }
}
```

你可能已经料到，这个设备会生成随机数字，并在得到偶数时返回第一个得到的偶数。该设备中有两个重载的 API 函数。¹ 一个函数永远不会超时，而另一个函数有毫秒级的超时时限。如果该设备没有在该时限内获得偶数，该函数就会返回 `Option` 类型的 `None` 值。显然，如果使用 `nextEvenNumber()` 函数的无超时时限版本，调用该函数的 `Actor` 对象所在的线程就会被阻塞，直到获得偶数为止。完成获取偶数的操作没有被定义时限。这种情况是绝对无法接受的，因此 `Actor` 对象必定会总是调用该函数的 `nextEvenNumber(waitFor: Int)` 版本。

下面是 `EvenNumberDevice` 对象用于设置时限的 `Timeout` 对象：

```
class Timeout(withinMillis: Int) {
    val mark = currentTime

    def isTimedOut(): Boolean = {
```

¹ 译者注：重载方法是指多个方法使用同一个名称，换言之，一个方法含有多个分支，每个分支执行不同的处理操作。

```

    if (withinMillis == -1) false
    else currentTime - mark >= withinMillis
  }

  private def currentTime(): Long = {
    (new Date()).getTime
  }
}

```

这个示例应用程序会处于等待状态，直到 `EvenNumberMonitor` 对象成功从 `EvenNumberDevice` 对象读取 10 个偶数为止。当该应用程序命令 `EvenNumberMonitor` 对象监听 `EvenNumberDevice` 对象时，`EvenNumberMonitor` 对象就会启动。

```

object DevicePollingConsumerDriver
  extends CompletableApp(10) {
    val evenNumberDevice = new EvenNumberDevice()

    val monitor = system.actorOf(
      Props(classOf[EvenNumberMonitor],
        evenNumberDevice),
      "evenNumberMonitor")

    monitor ! Monitor()

    awaitCompletion
  }

```

最后这段是 Actor 对象 `EvenNumberMonitor` 的实现代码：

```

case class Monitor()
class EvenNumberMonitor(
  evenNumberDevice: EvenNumberDevice)
  extends Actor {
    val scheduler =
      new CappedBackOffScheduler(
        500,
        15000,
        context.system,
        self,
        Monitor())

    def monitor = {
      val evenNumber = evenNumberDevice.nextEvenNumber(3)
      if (evenNumber.isDefined) {

```

```

println(s"EVEN: ${evenNumber.get}")
scheduler.reset
DevicePollingConsumerDriver.completedStep
} else {
println(s"MISS")
scheduler.backOff
}
}

def receive = {
case request: Monitor =>
monitor
}
}

```

这个起监听作用的 Actor 对象，在其内部创建了一个 CappedBackOffScheduler 实例。CappedBackOffScheduler 对象被用作以一定频率向 EvenNumberMonitor 发送 Monitor 消息的调度器。该频率最初被设置为最少 500 毫秒发送一次。如果检测到 EvenNumberDevice 对象生成了一个偶数，那么下次发送 Monitor 消息的时间间隔仍为 500 毫秒。否则，就会将下次发送 Monitor 消息的时间间隔增加一倍。下面是 CappedBackOffScheduler 对象的实现代码：

```

class CappedBackOffScheduler(
  minimumInterval: Int,
  maximumInterval: Int,
  system: ActorSystem,
  receiver: ActorRef,
  message: Any) {

  var interval = minimumInterval

  def backOff = {
    interval = interval * 2
    if (interval > maximumInterval)
      interval = maximumInterval
    schedule
  }

  def reset = {
    interval = minimumInterval
    schedule
  }

  private def schedule = {
    val duration =
      Duration.create(

```



```

        interval,
        TimeUnit.MILLISECONDS)

    system
        .scheduler
        .scheduleOnce(
            duration,
            receiver,
            message)
    }
}

```

每当这个调度器调用 `backOff` 方法后，该方法都会计算新的发送 `Monitor` 消息的时间间隔，直到该时间间隔到达 15 秒为止。当获得某个偶数，该调度器调用 `reset` 方法后，该时间间隔会被重新设置为 500 毫秒。不论通过上述哪种方式设置该时间间隔，该时间间隔都会被用于设置发送 `Monitor` 消息的频率。`Monitor` 消息会根据被设定的频率，被发送给 `EvenNumberMonitor` 对象。

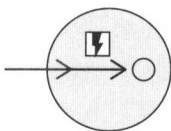
下面是 `EvenNumberMonitor` 对象的输出结果：

```

EVEN: 65290
EVEN: 67208
EVEN: 53130
MISS
MISS
EVEN: 63720
EVEN: 1810
EVEN: 25708
MISS
EVEN: 38840
EVEN: 92860
EVEN: 78328
EVEN: 87076

```

由事件驱动的消费者



`Actor` 模型中的 `Actor` 对象会自然而然成为由事件驱动的消费者，并实现点对点通道。因为这些 `Actor` 对象使用直接的异步消息传输模式，所以所有 `Actor` 对

象都会通过异步方式，处理由其他 Actor 对象发送给它们的消息，如图 9.4 所示。

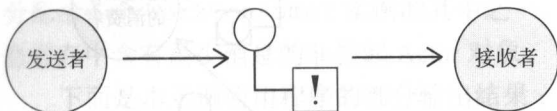
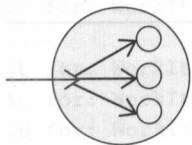


图 9.4 Actor 模型会自然而然地形成由事件驱动的消费者系统。

事件消息并不是使 Actor 对象成为由事件驱动的消费者的必要条件。事件消息可以由命令消息或文档消息替代。更确切地说，如果 Actor 对象收到任何类型的消息都会使之具有响应性，那么该 Actor 对象就是一个由事件驱动的消费者。因此，术语“由事件驱动的”的作用是与轮询做对比，以便描述消费者接收的消息的类型。

具有竞争性的消费者



具有竞争性的消费者能够同时对多条消息做出回应。根据轮询消费者和消息调度器的实现方式，会自然而然地生成具有竞争性的消费者。

图 9.5 展示了一个用于调度工作的消息调度器。这个工作调度器拥有许多处理工作的具有竞争性的消费者（图 9.5 展示了 3 个）。当该调度器收到指明有需要处理的工作的消息时，它就会将实际的工作任务分发给某个处理工作的消费者。那么它会将工作分发给哪个消费者呢？所有消费者都争相要获得工作，但当前负荷最少的消费者才能获得工作。实际上，如果能够通过某种方式确定多个消费者中的一个是完全空闲的，那么这个消费者就是最适合获得工作的消费者。

使用 Akka 标准路由器 `SmallestMailboxPool` 支持具有竞争性的消费者，可以取得特别好的效果。通过配置该路由器，可以使该路由器拥有含有多个消费者的池，从而使该路由器能够将消息发送给消息缓存中含有最少消息的非挂起消费者。这意味着所有消费者 Actor 对象都拥有自己的消息缓存。消费者池中含有的消费者既可以是本地 Actor 对象也可以是远程 Actor 对象，但是使用远程消费者 Actor 对象的效果较差，因为 `SmallestMailboxPool` 路由器无法查看它的消息缓存中含有多少消息。由此可见，选择向远程消费者 Actor 对象发送消息，会是该路由器迫不得已的最后选项。

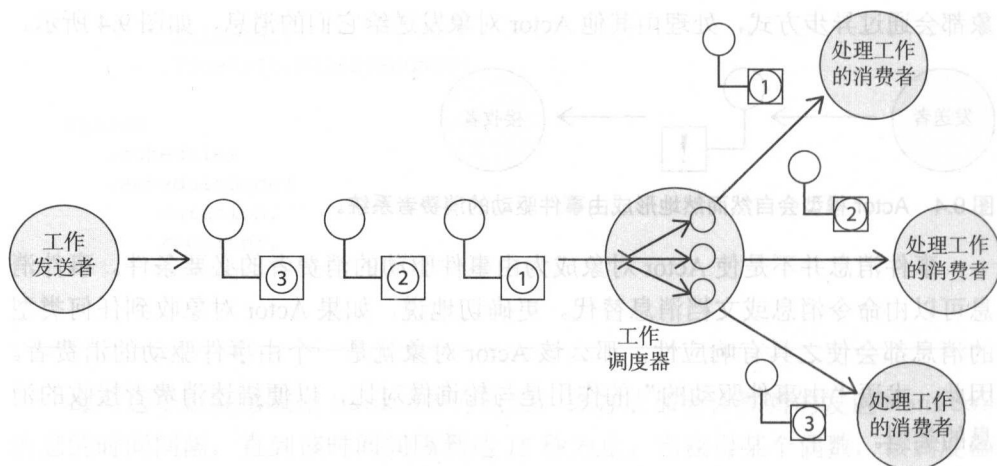


图 9.5 具有竞争性的消费者会根据它们消息缓存中含有的消息数量，选择是否接收新的工作。

下面是一个使用 `SmallestMailboxPool` 路由器的例子：

```
object CompetingConsumerDriver
  extends CompletableApp(100) {
    val workItemsProvider =
      system.actorOf(
        Props[WorkConsumer]
          .withRouter(
            SmallestMailboxPool(
              nrOfInstances = 5)))

    for (itemCount <- 1 to 100) {
      workItemsProvider ! WorkItem("WorkItem" + itemCount)
    }

    awaitCompletion
  }

case class WorkItem(name: String)

class WorkConsumer extends Actor {
  def receive = {
    case workItem: WorkItem =>
      println(s"${self.path.name} for: ${workItem.name}")

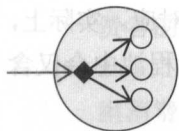
    CompetingConsumerDriver.completedStep
  }
}
```

当由 `workItemsProvider` 变量引用的 `Actor` 对象收到消息时，就会将工作分配给 5 个 `WorkConsumer` 实例的其中之一。分配工作的原则是将工作分配给消息缓存中含有最少消息的非挂起 `Actor` 对象。

下面是本示例应用程序的部分输出结果：

```
$a for: WorkItem1
$c for: WorkItem3
$a for: WorkItem6
$d for: WorkItem4
$a for: WorkItem11
$d for: WorkItem9
$a for: WorkItem16
$d for: WorkItem14
$a for: WorkItem21
$d for: WorkItem19
$d for: WorkItem24
$b for: WorkItem2
$c for: WorkItem8
...
$b for: WorkItem98
$c for: WorkItem99
$d for: WorkItem100
$e for: WorkItem91
$a for: WorkItem90
```

消息调度器



消息调度器与基于内容的路由器类似。消息调度器会查看消息的内容（如消息类型），然后将消息分发给与消息类型匹配的 `Actor` 对象。然而，基于内容的路由器不会关心 `Actor` 对象的工作负荷，而工作负荷通常是消息调度器关注的因素。而且，基于内容的路由器经常跨越处理过程分发消息，而消息调度器常常在同一个处理过程中分发工作。即使在将工作分配给特定类型的消费者前，消息调度器会检查某些消息内容（如消息类型），但它更关注的是它消费者池中的消费者的工作负荷，并且会将工作分配给工作负荷最轻的消费者。图 9.6 展示了这种分配规则。

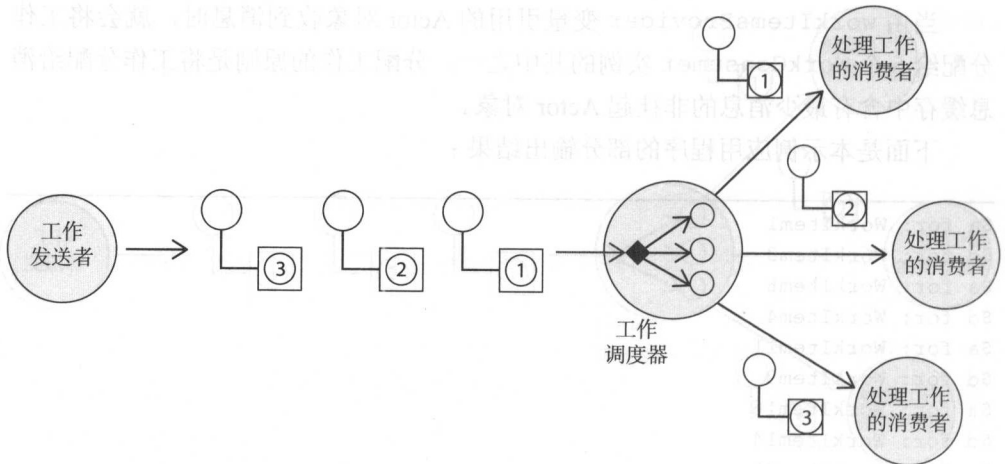


图 9.6 消息调度器是一种工作调度器。

Akka 框架提供了各种消息调度器（实际上还提供了一些路由器），在消息调度器模式中使用这些 Akka 消息调度器可以取得很好的效果。同时应用消息调度器和路由器的情况是由于历史原因导致的。在路由器出现前，负载均衡调度器（如 `BalancingDispatcher` 对象）就已经出现了。然而，`BalancingDispatcher` 对象有一点难用，因此 Akka 开发团队决定使用路由器取代它。有些人认为如果这个决策是现在做出的，那么 `BalancingDispatcher` 对象就不会出现。

下面是一些标准的 Akka 消息调度器。

- `Dispatcher`: 默认的调度器，能够将线程池与一系列 Actor 对象关联起来。你可以根据具体的 Actor 对象选择不同的调度器。
- `PinnedDispatcher`: 该调度器会将线程与 Actor 对象一一对应。实际上，在使用这种调度器时，Actor 对象仍旧拥有线程池，但该线程池中会仅含有一个线程。
- `BalancingDispatcher`: 该调度器会将消息分发给工作负荷最轻的 Actor 对象（如完全空闲的 Actor 对象）。该调度器管理的所有 Actor 对象都共用一个消息缓存，而且这些 Actor 对象的类型必须相同（现在不推荐使用这种调度器）。

下面的代码创建了拥有 5 个 Actor 对象 `WorkConsumer` 的负载均衡调度器，该调度器会为这些 Actor 对象分发 100 条消息：

```
val workConsumers: List[ActorRef] =
```

```

for (otherWorks <- (1 to 5).toList) yield {
  system.actorOf(Props[WorkConsumer]
    .withDispatcher("workconsumer-dispatcher"))
}

val bestWorkConsumer = workConsumers(0)

for (itemCount <- 1 to 100) {
  bestWorkConsumer ! WorkItem("WorkItem" + itemCount)
}

```

尽管该调度器管理着 5 个 WorkConsumer 实例，但同一时刻它只能为一个 WorkConsumer 实例分发工作。这是因为这 5 个 WorkConsumer 实例共用了同一个消息缓存。当向最适合接收消息的 WorkConsumer 对象发送消息时，消息会被发送给这 5 个对象中工作负荷最轻的一个。你的 Akka 配置必然会用到 workconsumer-dispatcher 设置：

```

workconsumer-dispatcher {
  type = BalancingDispatcher
}

```

尽管可以使用 BalancingDispatcher 调度器，但不建议你使用它。如果你选择使用 Akka 标准工具，最好选择一种路由器。下面列出了 Akka 框架中的标准路由器。尽管它们被称为路由器，但实际上它们是非常优秀的消息调度器。

- RoundRobinRouter：通过配置该路由器，可以使该路由器管理一组消费者，并以循环方式向这些消费者分发消息。这意味着该路由器不会关心消费者的工作负荷。
- SmallestMailboxRouter：通过配置该路由器，可以使该路由器管理一组消费者，并向消息缓存中含有最少消息的非挂起消费者发送消息。这意味着所有消费者都拥有自己的消息缓存。这些消费者既可以是本地 Actor 对象也可以是远程 Actor 对象，但使用远程 Actor 对象的效果较差，因为该路由器无法查看远程 Actor 对象的消息缓存。由此可见，选择向远程消费者 Actor 对象发送消息，会是该路由器迫不得已的最后选项（这也许是该路由器替代 BalancingDispatcher 调度器的最佳理由，你可以参阅前面介绍具有竞争性的消费者的示例）。

下面是拥有 5 个 WorkConsumer 对象（代表消费者）的 RoundRobinRouter 路由器的代码：

```

val workItemsProvider = system.actorOf(
  Props(classOf[WorkConsumer], workItemsProvider)
    .withRouter(
      RoundRobinRouter(
        nrOfInstances = 5)))

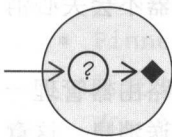
workItemsProvider ! WorkItem("WorkItem1")
workItemsProvider ! WorkItem("WorkItem2")
workItemsProvider ! WorkItem("WorkItem3")
workItemsProvider ! WorkItem("WorkItem4")
workItemsProvider ! WorkItem("WorkItem5")

```

本例中的每条 `WorkItem` 消息，都会被发送给不同的 `WorkConsumer` 对象。请参阅前面介绍具有竞争性的消费者的内容，查看使用 `SmallestMailboxRouter` 路由器的示例。

除了使用标准的 Akka 调度器和路由器，还可以使用前面介绍过的轮询消费者模式。轮询消费者模式允许任何消费者 Actor 对象，在需要获取工作时，请求工作提供者提供工作。这要求消费者知道工作提供者的地址。而且因为消费者必须向工作提供者发送要求获取工作的消息，所以还需要发送更多消息。但轮询消费者模式节省了使用 Akka 标准工具时需要花费的检查系统开销，而且也不要求你深入了解 Akka 框架。

选择性消费者



如果你设计的 Actor 对象会收到各种类型的消息，但是只能处理其中的一部分，就应将该 Actor 对象设计成选择性消费者。在这种情况下，选择性消费者是一种消息过滤器，仅允许系统处理它支持的消息。请参阅介绍消息过滤器的内容，并与选择性消费者做对比。

还可以使选择性消费者代表数据类型消费者接收各种类型的消息，在这种模式中选择性消费者会将各种类型的消息发送到数据类型通道中。第 7 章介绍过这种模式，下面介绍一个不使用动态路由规则的示例。

下面的示例应用程序 `SelectiveConsumerDriver` 创建了 3 个 Actor 对象，

每个 Actor 对象都专门处理一种消息，从而使它们形成了数据类型通道。

```
object SelectiveConsumerDriver
  extends CompletableApp(3) {
    val consumerOfA =
      system.actorOf(
        Props[ConsumerOfMessageTypeA],
        "consumerOfA")

    val consumerOfB =
      system.actorOf(
        Props[ConsumerOfMessageTypeB],
        "consumerOfB")

    val consumerOfC =
      system.actorOf(
        Props[ConsumerOfMessageTypeC],
        "consumerOfC")

    val selectiveConsumer =
      system.actorOf(
        Props(new SelectiveConsumer(
          consumerOfA, consumerOfB, consumerOfC)),
        "selectiveConsumer")

    selectiveConsumer ! MessageTypeA()
    selectiveConsumer ! MessageTypeB()
    selectiveConsumer ! MessageTypeC()

    awaitCompletion
  }
```

创建好这 3 个消费者 Actor 对象后，该程序创建了 SelectiveConsumer 对象并发送了 3 条消息：MessageTypeA、MessageTypeB 和 MessageTypeC。这些消息会被 SelectiveConsumer 对象收到，然后被分发给数据类型通道。

```
case class MessageTypeA()
case class MessageTypeB()
case class MessageTypeC()

class SelectiveConsumer(
  consumerOfA: ActorRef,
  consumerOfB: ActorRef,
  consumerOfC: ActorRef) extends Actor {
```

```

def receive = {
  case message: MessageTypeA =>
    consumerOfA forward message
  case message: MessageTypeB =>
    consumerOfB forward message
  case message: MessageTypeC =>
    consumerOfC forward message
}
}

```

下面是专门处理特定类型消息的消费者：

```

class ConsumerOfMessageTypeA extends Actor {
  def receive = {
    case message: MessageTypeA =>
      println(s"ConsumerOfMessageTypeA: $message")
      SelectiveConsumerDriver.completedStep
  }
}

class ConsumerOfMessageTypeB extends Actor {
  def receive = {
    case message: MessageTypeB =>
      println(s"ConsumerOfMessageTypeB: $message")
      SelectiveConsumerDriver.completedStep
  }
}

class ConsumerOfMessageTypeC extends Actor {
  def receive = {
    case message: MessageTypeC =>
      println(s"ConsumerOfMessageTypeC: $message")
      SelectiveConsumerDriver.completedStep
  }
}

```

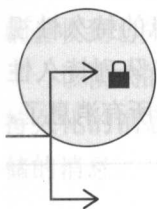
运行该示例应用程序可以获得下面的输出结果：

```

ConsumerOfMessageTypeA: MessageTypeA ()
ConsumerOfMessageTypeC: MessageTypeC ()
ConsumerOfMessageTypeB: MessageTypeB ()

```

持久订阅者



当需要确保接收者 Actor 对象在没有被激活的情况下，不会错过已发送的消息时，可使用持久订阅者。在我撰写本书时，Akka 工具集为持久订阅者提供了确切的支持。然而，如果你需要使用这些工具，尽可以自由地使用它们，不必担心它们会脱离你的掌控。而且，尽管 *Enterprise Integration Patterns* 一书中提出只应在发布—订阅通道中使用这种模式，但实际上你可以在任何类型的 Actor 消息通道中使用该模式。

Akka Persistence 插件支持持久性视图模式。持久性视图是 Akka 框架支持 CQRS 模式的一个途径 [CQRS]，在这种模式中，持久性 Actor 对象能够将它状态投射到自定义的视图中。你可以使用相同的机制创建持久订阅者。

该解决方案的第一个部分是，创建用于代表特殊消息通道的持久性 Actor 对象，消息会通过该对象发布或执行入队操作。可以将这种持久性 Actor 对象视为拥有名称的、接收被发布 / 发送消息的主题 / 队列。持久性 Actor 对象的名称是由该对象的 `persistenceId` 字段（代表具有唯一性的标识符）提供的。当拥有名称的主题 / 队列持久性 Actor 对象收到消息时，该 Actor 对象做的全部事情仅是持久化消息。

```
class PublishedTopic extends PersistentActor {

  override def persistenceId = "my-topic-name"

  override def receiveCommand: Receive = {
    case message: Any =>
      persistAsync(message)
  }
}
```

在这类情况中，可以使用 `persist()` 方法的特殊版本：`persistAsync()`。不必在成功执行完持久化操作后才接收下一条消息。因此，可以对持久性 Actor 对象使用最优化的持久化操作。

该解决方案的第二个部分是，为每个订阅者创建一个持久性视图实例。如果你想像处理拥有名称的队列一样处理该对象，那么只能将一个持久性视图用作队列消费者。在任何情况中，从主题/队列持久性 Actor 对象接收消息的持久性视图 Actor 实例，都会将它的 `persistenceId` 字段设置为相应主题/队列持久性 Actor 对象标识符。这样它就能够接收被拥有名称的主题/队列持久化的所有消息了。

```
class TopicSubscriber extends PersistentView {
```

```
  override def persistenceId = "my-topic-name"
  override def viewId = "my-topic-name-subscriber-1"
  var dummyState = 0
```

```
  def handleMessage(message: Any) = {
    ...
  }
```

```
  override def receive: Receive = {
    case message: Any =>
      handleMessage(message)
      deleteSnapshots(SnapshotSelectionCriteria.Latest)
      saveSnapshot(dummyState)
  }
```

```
  override def receiveRecover: Receive = {
    case SnapshotOffer(metadata, snapshot) =>
      dummyState = snapshot // 实际上可以丢弃
    case RecoveryCompleted =>
  }
}
```

注意

在我撰写本书时，持久性视图模式正要被 Akka Streams 解决方案取代。我可以确定 Akka Streams 解决方案不会比持久性视图模式差，而且很直观和简单易用。

在使用 TopicSubscriber 对象时还有一个附加要求，该对象必须将它已经收到的消息记录下来。通过在收到消息时生成新的快照，该对象做到了这一点，如上面的示例程序所示。这样当 TopicSubscriber 对象被停止并重启后，就不会重复处理已经收到过的消息，而仅会处理它之前没有收到过的消息。

通过这种设置，即使在 TopicSubscriber 对象不运行的时候，也永远不会错过发送给 PublishedTopic 对象的消息。当 TopicSubscriber 对象重启后，就会收到它先前没有收到的消息。

该模式的一个缺点是，无法删除被持久化 Actor 对象（如 PublishedTopic）持久化的消息。必要时，可在确定安全的情况下，定期删除持久化 Actor 对象存储的消息。

```
class PublishedTopic extends PersistentActor {
  override def persistenceId = "my-topic-name"

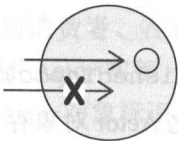
  override def receiveCommand: Receive = {
    case cg: CollectGarbage =>
      deleteMessages(cg.maximumSequenceNr)
    case message: Any =>
      persist(message)
  }
}
```

这段代码会将持久化 Actor 对象存储的消息中，序号小于等于 maximumSequenceNr 值的消息永久删除（但是，只有当基础存储机制支持该功能时，才能真正执行永久删除操作）。你需要仔细考虑怎样才能安全地删除一系列消息。例如，你可以使用基于订阅者确认的方式。然而，你可能不知道有多少个订阅者，因此这种方式不实用。还可以根据时间日期确定是否安全，这实际上就是确定每条消息的最长保留时间。当然，如果你的持久化 Actor 对象仅有一个监听者或者你能够确定其监听者的数量，那么这个问题就很好解决。归根结底，如果你不必过于担心持久化消息的清除问题，那么使用该模式就可以取得最佳效果。

另一个完全不同的选项是，围绕 Apache Kafka 服务（其最初的开发目标是成为一种高吞吐量的发布—订阅机制）创建解决方案 [Kafka]。该解决方案比上面介绍的解决方案更加成熟，而且拥有所有发布—订阅功能（包括内置的消息缓存上限功能）。

也许 Akka 开发团队或其他愿意为 Akka 框架做贡献的程序员会开发出更强大的持久订阅者解决方案，但在此之前，此处介绍的解决方案可以为你提供无须付出额外代价的正确语义。如果你想使程序设计接口更像发布—订阅通道，可通过使用了正确术语的特征，封装持久性 Actor 对象和持久性视图。

幂等接收者



当某个 Actor 对象可能会多次接收同一条消息，而且每次处理这条消息都会出问题时，可以将该 Actor 对象设计为幂等接收者。因为 Actor 模型中的标准消息传输协定是至多发送一次，所以好像永远不会出现重复发送某一条消息的问题。但当出现下列情况时，该问题还是会出现：

消息发送者希望能够收到一份回执，如果它没有收到这份回执，就会再次发送同一条消息。接收者收到并处理了这条消息，但它的回执并没有被送达发送者。如果接收者没有被设计为幂等接收者，重复接收已经被处理过的消息会导致 Actor 对象出现状态方面的问题，而且会影响处理接收者发出的已处理消息的其他 Actor 对象。

实际上，这种情况不仅会导致一个或多个 Actor 处于非法状态，而且会影响系统中大部分组成部分的状态。不论是使用了确保送达机制还是使用了持久订阅者，这种情况都会出现，因为至多发送一次的传送机制被替换为至少发送一次传送机制。

可以通过几种方式避免出现这种情况，并将 Actor 对象设计为幂等接收者。将 Actor 对象设计为幂等接收者，意味着当 Actor 对象重复接收同一条消息时，其状态不会发生改变。下面列出了这几种方式：

- 避免处理消息副本，通过 id 识别出消息副本。
- 设计状态切换消息，使 Actor 对象在每次收到这类消息时都执行相同的切换操作。
- 使状态切换操作能够不受收到的消息副本的影响。

避免处理消息副本

Enterprise Integration Patterns 一书详细介绍过避免处理消息副本的方式。要实现这种方式，需要消息接收者记录它已经收到的消息，并忽略被重复发送的消息。可以将已收到消息的 id 缓存在内存中，也可以将它们保存到数据库中。这意味着每条非副本消息都必须被赋予一个具有唯一性的 id。*Enterprise Integration Patterns* 一书中警告：不应将业务标识符用作非副本消息的标识符，因为多条消

息可能会拥有相同的业务标识符。

使用避免处理消息副本方式时需要应对的另一个问题是，确定保留已收到消息 id 的时限。*Enterprise Integration Patterns* 一书建议使用传输控制协议 /Internet 协议 (TCP/IP) 避免重复处理报文的方式 (使用窗口尺寸)。可以将消息 id 记录的数量设置为 100 或 1,000 条，当到达这个窗口尺寸时，最早的消息 id 记录会被丢弃。当然，如果窗口尺寸设置得过小会带来危险，因此应谨慎地设置该功能。基于时间的窗口尺寸可能会比使用固定集合的窗口尺寸更好。

使消息具有相同的效果

如 *Enterprise Integration Patterns* 一书中所述，你可以将消息设置为每次被 Actor 对象处理后，都会对 Actor 对象的状态产生相同的影响。下面是 *Enterprise Integration Patterns* 使用过的一个示例，它比较了两种类型的消息和它们的使用方式：

```
case class Deposit(amount: Money)
...
account ! Deposit(Money(10))
...
case class SetBalance(amount: Money)
...
account ! SetBalance(currentBalance + Money(10))
```

第一类消息 `Deposit` 必须通过某种方式避免该类消息被重复处理。否则，如果同一条消息被接收两次，就会使账号中的资金数量出错。不必对第二类消息 `SetBalance` 使用避免重复处理机制，因为这类消息仅用于设置指定的状态。

我非常不喜欢这种处理方式，尤其是本例展示的这个方式。在任何消息传输环境中，要在查询当前状态的操作和收到切换到下一个状态的命令消息之间，使接收者不改变其状态几乎是不可能的。这不是说无法设计出实现该目的的消息类型，而是说我认为本例从根本上就无法接受这种方式。因此，可以将这个例子做如下修改：

```
case class Deposit(
  transactionId: TransactionId,
  amount: Money)
...
account ! Deposit(TransactionId(), Money(10))
```

令人奇怪的是，为了修改这个示例，我们遇到了第三种处理方式。

使状态切换操作不受收到消息副本的影响

让我们继续处理上一节使用的示例，使用 `TransactionId` 字段（代表 Actor 对象处理事务的 id）使 Actor 对象在收到 `Deposit` 消息（代表存入资金）的副本时不受影响。是否必须使用 `TransactionId` 字段处理消息副本呢？虽然这个问题的答案不是肯定的，但是这种方式会使解决方案变得非常简单：

```
case class AccountBalance(
  accountId: AccountId,
  amount: Money)

case class Deposit(
  transactionId: TransactionId,
  amount: Money)

case class QueryBalance()

case class Withdraw(
  transactionId: TransactionId,
  amount: Money)

class Account(accountId: AccountId) extends Actor {
  val transactions = Map.empty[TransactionId, Transaction]

  def receive = {
    case deposit: Deposit =>
      val transaction =
        Transaction(
          deposit.transactionId,
          deposit.amount)
      println(s"Deposit: $transaction")
      transactions += (deposit.transactionId -> transaction)
      AccountDriver.completedStep
    case withdraw: Withdraw =>
      val transaction =
        Transaction(
          withdraw.transactionId,
          withdraw.amount.negative)
      println(s"Withdraw: $transaction")
      transactions +=
        (withdraw.transactionId -> transaction)
      AccountDriver.completedStep
    case query: QueryBalance =>
      sender ! calculateBalance()
      AccountDriver.completedStep
  }
}
```

```

}

def calculateBalance(): AccountBalance = {
  var amount = Money(0)

  transactions.values map { transaction =>
    amount = amount + transaction.amount
  }

  println(s"Balance: $amount")

  AccountBalance(accountId, amount)
}

```

因为 transactions 变量代表的 Map 映射，将一次处理事务的操作与一个具有唯一性的 TransactionId 值对应了起来，所以即使多次收到同一条 Deposit 消息也不会对 Account 对象（代表账号）中的资金产生有害影响。在这种情况下，因为特殊的 Map 操作本身具有幂等性，所以成功避免了消息副本对 Actor 对象产生影响。要精确地计算出账号中的资金数量，只需添加所有 Deposit 消息代表的数额并减去所有 Withdraw 消息代表的数额。

下面是该示例的代码和输出结果：

```

object AccountDriver extends CompletableApp(17) {
  val account =
    system.actorOf(
      Props(classOf[Account], AccountId()),
      "account")

  val deposit1 = Deposit(TransactionId(), Money(100))

  account ! deposit1
  account ! QueryBalance()
  account ! deposit1
  account ! Deposit(TransactionId(), Money(20))
  account ! QueryBalance()
  account ! deposit1
  account ! Withdraw(TransactionId(), Money(50))
  account ! QueryBalance()
  account ! deposit1
  account ! Deposit(TransactionId(), Money(70))
  account ! QueryBalance()
  account ! deposit1
  account ! Withdraw(TransactionId(), Money(100))
  account ! QueryBalance()
}

```

```

account ! deposit1
account ! Deposit(TransactionId(), Money(10))
account ! QueryBalance()

awaitCompletion
}

Deposit: Transaction(TransactionId(1), Money(100.0))
Balance: Money(100.0)
Deposit: Transaction(TransactionId(1), Money(100.0))
Deposit: Transaction(TransactionId(2), Money(20.0))
Balance: Money(120.0)
Deposit: Transaction(TransactionId(1), Money(100.0))
Withdraw: Transaction(TransactionId(3), Money(-50.0))
Balance: Money(70.0)
Deposit: Transaction(TransactionId(1), Money(100.0))
Deposit: Transaction(TransactionId(4), Money(70.0))
Balance: Money(140.0)
Deposit: Transaction(TransactionId(1), Money(100.0))
Withdraw: Transaction(TransactionId(5), Money(-100.0))
Balance: Money(40.0)
Deposit: Transaction(TransactionId(1), Money(100.0))
Deposit: Transaction(TransactionId(6), Money(10.0))
Balance: Money(50.0)

```

尽管第一条 Deposit 消息被重复发送了 6 次，但只有当 Account 对象第一次收到它时才会执行存入资金的事务。

使 Actor 对象拥有幂等性的第二种方式是，保持一种允许或禁止执行特定状态切换操作的状态。可以将上一个例子中的 Map 事务映射，用作状态或指明是否可执行特定操作的标记。

```

class Account(accountId: AccountId) extends Actor {
  val transactions = Map.empty[TransactionId, Transaction]

  def receive = {
    case deposit: Deposit
      if (!transactions.contains(
        deposit.transactionId)) =>
      val transaction =
        Transaction(deposit.transactionId, deposit.amount)
      println(s"Deposit: $transaction")
      transactions += (deposit.transactionId -> transaction)
      AccountDriver.completedStep
    ...
  }
}

```

将这个条件表达式添加到 Deposit 消息的匹配语句中后，如果 Deposit 消息中的 TransactionId 值已经被添加到 Map 映射中，那么这条 Deposit 消息就不会被处理。你可以通过许多方式设计可变（易变）状态从一个状态切换到另一个状态的方式，以及使用各种状态代表 Actor 对象已经收到一条或多条消息。反过来，如果 Actor 对象处于某个状态，你就能知道一条或多条消息已经被处理。

```
case attachment: AttachDocument
  if (document.isNotAttached) =>
    ...
```

还可以通过另一种方式确保 Actor 对象在收到消息副本时具有状态安全性。这种方式也是以状态为基础的，但是使用了不同的切入角度。这种方式被称为变成（become），下面使用内置的 Akka 框架 Actor 功能，通过这种方式可管理风险评估分类。

```
case class AttachDocument(documentText: String)
case class ClassifyRisk()
case class RiskClassified(classification: String)
case class Document(documentText: Option[String]) {
  if (documentText.isDefined) {
    val text = documentText.get
    if (text == null || text.trim.isEmpty) {
      throw new IllegalStateException(
        "Document must have text.")
    }
  }
}

def determineClassification = {
  val text = documentText.get.toLowerCase

  if (text.contains("low")) "Low"
  else if (text.contains("medium")) "Medium"
  else if (text.contains("high")) "High"
  else "Unknown"
}

def isNotAttached = documentText.isEmpty
def isAttached = documentText.isDefined
}

class RiskAssessment extends Actor {
  var document = Document(None)
```

```

def documented: Receive = {
  case attachment: AttachDocument =>
    // 已经接收过; 忽略

  case classify: ClassifyRisk =>
    sender ! RiskClassified(
      document.determineClassification()
    )
}

def undocumented: Receive = {
  case attachment: AttachDocument =>
    document = Document(Some(attachment.documentText))
    context.become(documented)
  case classify: ClassifyRisk =>
    sender ! RiskClassified("Unknown")
}

def receive = undocumented
}

```

让我们观察两个偏函数：documented 和 undocumented。这些偏函数代表 RiskAssessment 对象用于接收消息的代码块。该接收代码块最初被设置为 undocumented 偏函数。当 undocumented 代码块收到 AttachDocument 消息（代表添加风险评估文档的命令）时，Document 对象（代表风险评估文档）会被创建并会根据 RiskAssessment 对象（代表风险评估器）的状态被设置。同时，RiskAssessment 对象会命令它的接收消息代码块变为 documented 偏函数。

```

case attachment: AttachDocument =>
  document = Document(Some(attachment.documentText))
  context.become(documented)

```

还应注意，当将接收消息的代码块设置为 undocumented 偏函数时，所有视图获得风险分类文档的操作，都会得到 RiskClassified("Unknown") 结果（代表未收到任何文档消息）。

当 RiskAssessment 对象的接收消息代码块变为 documented 偏函数时，表明由于已经收到过 AttachDocument 消息，该对象的状态已经切换，因此该 Actor 对象的 documented 代码块现在可以安全地忽略被重复发送的 AttachDocument 消息了。此外，只有当该 Actor 对象的接收消息代码变为 documented 偏函数时，才能对获取 ClassifyRisk 消息（代表对风险评估文档进行分类的命令）的操作做出回应。本例的风险评估分类操作非常简单，仅通过文档文本中是

否含有单词 low、medium 或 high，判定风险评估文档的类别。

让我们先看下面的示例应用程序，该程序使用 Akka 框架中的 Future 对象获取问题的答案：

```
object RiskAssessmentDriver extends CompletableApp(2) {
  implicit val timeout = Timeout(5 seconds)

  val riskAssessment =
    system.actorOf(
      Props[RiskAssessment],
      "riskAssessment")

  val futureAssessment1 = riskAssessment ? ClassifyRisk()
  printRisk(futureAssessment1)

  riskAssessment ! AttachDocument("This is a HIGH risk.")

  val futureAssessment2 = riskAssessment ? ClassifyRisk()
  printRisk(futureAssessment2)

  awaitCompletion

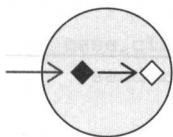
  def printRisk(futureAssessment: Future[Any]): Unit = {
    val classification =
      Await.result(futureAssessment, timeout.duration)
        .asInstanceOf[RiskClassified]
    println(s"$classification")
    completedStep
  }
}
```

请注意，当该程序被要求提供风险评估分类信息时，会给出下列答案。

```
RiskClassified(Unknown)
RiskClassified(High)
```

这样通过 Akka 框架支持的变成处理方式，我们成功将 RiskAssessment 对象设计成了一个幕等接收者。

服务激活剂



当内部服务需要使用一个或多个外部资源时，可使用服务激活剂。例如，客户端可能会通过 Actor 对象、中间件消息传输系统、RESTful 资源或远程过程调用（RPC），执行请求操作。不论通过怎样的方式执行该请求操作，该请求的接收者都会委托内部的应用程序服务满足该请求，如图 9.7 所示。

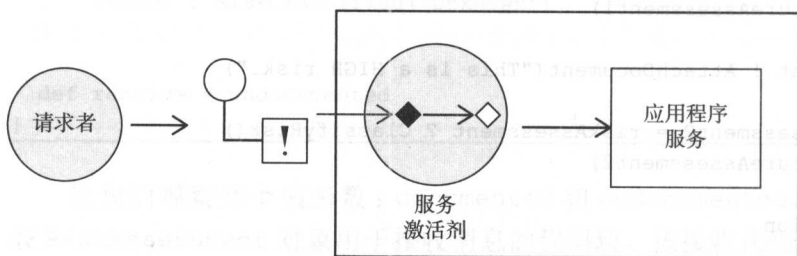


图 9.7 服务激活剂将请求委托给应用程序服务处理。

这符合 *Implementing Domain-Driven Design* 一书中提出的六角形或端口和适配器架构，如图 9.8 所示。客户端应用程序只是与接收者应用程序的外部适配器进行交互，然后外部适配器将满足请求的任务委托给接收者应用程序的内部组件（即应用程序服务）。

该架构能够在其外部提供各种类型的服务激活剂，同时支持其内部的、通用的、可重用的应用程序服务层。如图 9.8 所示，所有连入型适配器（图 9.8 左侧的）都是服务激活剂。这些类型服务激活剂都能够接收和传递给内部应用程序的 API 的参数（消息属性）。当然，如果应用程序使用响应式吞吐模式，可以将应用程序服务实现为 Actor 对象。

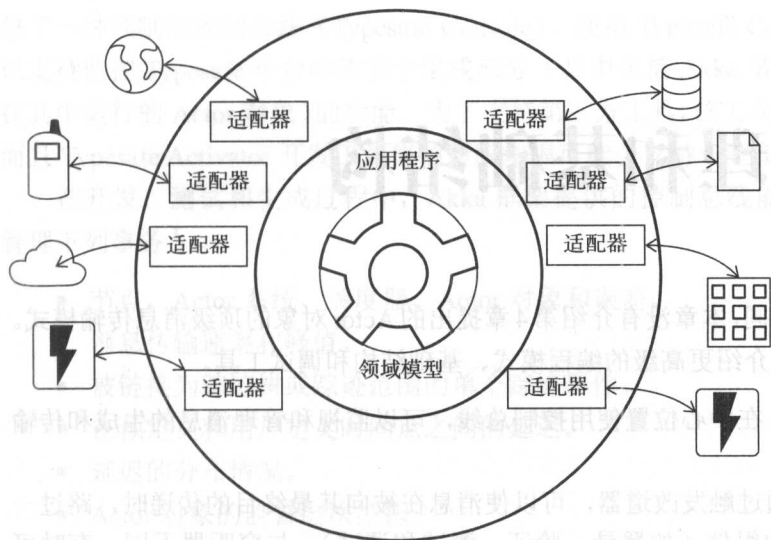


图 9.8 端口和适配器架构。

小结

本章详细介绍了十多种消息端点，其中包括两种在 Actor 对象中支持事务的消息端点。本章介绍了 Actor 模型提供的消息传输网关和实现消息传输映射的方式。通过事务型 Actor 对象使用事件溯源模式，支持聚合操作的方式。

本章还介绍了创建非阻塞式轮询消费者、定义由事件驱动的消费者，以及使用 Akka 标准路由器创建具有竞争性的消费者的方式。通过消息调度器实现负载均衡的重要性。将 Actor 对象设置为仅回应指定类型消息的选择性消费者的方式。使用 Akka Persistence 插件支持持久订阅者的方式。在使用确保送达机制时，将 Actor 对象设置为幂等接收者的必要性。最后，本章介绍了在应用程序的外部边缘，被设置为服务激活剂的 Actor 对象，以及服务激活剂从外部接收消息并将这些消息分发给应用程序内部的服务或领域模型 Actor 对象的方式。

第10章

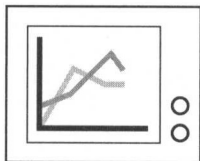
系统管理和基础结构

与前面各章不同,本章没有介绍第4章提出的 Actor 对象的顶级消息传输模式。更确切地说,本章介绍更高级的编程模式、基础结构和调试工具。

- **控制总线** : 在中心位置使用控制总线,可以监视和管理消息的生成和传输模式。
- **改道器** : 通过触发改道器,可以使消息在被向其最终目的传递时,路过一系列特殊的组件(如登录、验证、测试和调试)。与窃听器不同,有时可使用改道器在转发消息前修改消息的内容。
- **窃听器** : 使用窃听器可以检查在 Actor 对象之间传输的消息。
- **消息元数据 / 历史记录** : 通过在消息中添加元数据,可以表明谁对消息执行了哪些操作。
- **消息日志 / 存储器** : 使用消息日志可以支持确保送达机制和持久订阅者。
- **智能代理** : 有时需要设计窃听器,以便了解使用原始发送者的返回地址实现请求一回复模式的完整过程。使用智能代理可以完成这项任务。
- **测试消息** : 当需要查明消息是否已经被妥善接收时,可向 Actor 对象发送测试消息。
- **通道净化器** : 使用通道净化器可以在测试或继续生成消息前,从消息存储器中删除一些消息。

在企业级响应式应用程序中,消息元数据和消息日志最为常用。你可根据需要在不同的开发、测试和生产阶段,使用和组合使用这些工具。

控制总线



使用控制总线可以监视和管理 Actor 系统和其中的 Actor 对象。Akka 框架提

供了一种预制的控制总线 (Typesafe Console)。使用 Typesafe Console 控制总线可以支持监控 Typesafe 平台中的多个组成部分 (其中包括 Akka 节点、Actor 系统和在其中运行的 Actor 对象) 的功能。为了支持第三方工具, 该工具现在已经被弃用, 而且 Typesafe Activator 开发工具集的新版本很可能会支持一些重要的监控功能。

在开发、测试和生成过程中, Akka 框架提供的控制总线能够帮助你观察和管理下列事务¹:

- 节点、Actor 系统、调度器、Actor 对象和误差。
- 消息传输速率和峰值。
- 被链接为踪迹树或踪迹范围的单个踪迹事件。
- 在预定义和用户定义时间点之间的延迟。
- 延迟的分布情况。
- Actor 对象的监督层次结构。
- Actor 对象消息缓存中的队列尺寸和延迟情况。
- Actor 消息调度器的状态。
- 远程消息传输状态和系统错误。
- Java 虚拟机和操作系统的健康情况。

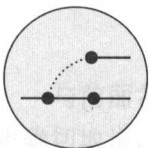
应根据节点、Actor 对象的分组或单个 Actor 对象, 划分被监控的数据。

- **节点**: 节点与 JVM 一一对应, 一台物理或虚拟服务器上可能会运行多个节点。你应该获取在全部托管的节点中, 选中某个节点从而查看其细节的能力。
- **节点概况**: 你应该获取在某个节点中查看其中正在运行的 Actor 系统的数量和单个 Actor 系统的细节的能力。
- **Actor 系统**: 查看在指定节点中正在运行的 Actor 系统。你应该在 Actor 系统中获取查看消息调度器数量、Actor 对象数量、误差数量, 以及消息调度器和 Actor 对象的健康情况的能力。
- **调度器**: 一个 Actor 系统中会含有一个或多个调度器。每个调度器都为多个 Actor 对象和误差服务。通过查看调度器的细节, 你就能够详细了解吞吐量、线程数量、执行操作的线程、线程池、消息缓存尺寸、消息计数和误差的情况。
- **Actor 对象**: 你应该获取查看在指定 Actor 系统中正在运行的 Actor 对象的能力。

¹ Typesafe Console 开发工具集的官方文档介绍了这些内容的一部分。

- 误差：你必须获取在指定 Actor 系统中，查看错误、警告、死信消息、死锁和未处理消息的能力。

改道器



通过使用改道器可以在 Actor 对象之间传输消息的过程中，通过上下文路由使消息经过一系列特殊的组件。在使用改道器时，可以根据需要设计特殊上下文路由的开关功能。使用改道器可以通过无缝方式，为 Actor 系统添加登录、验证、测试和调试功能。

与窃听器不同，改道器可以检查和修改消息的内容，而窃听器只能用于检查消息的内容。改道器通常会与控制总线关联，因为控制总线需要通过某种方式检查系统的各个方面，其中包括性能和错误。

仔细思考改道器的基础设计思路后，你就能够看清改道器实际上是管道和过滤器架构的一种形式。在 Actor 模型中，使用管道和过滤器实现改道器是一种特别高效的方式。实际上，如第 4 章举的管道和过滤器示例所示，使用一个或多个专用组件创建改道器的最简单方式之一，是将过滤器 Actor 对象与所有发送者 Actor 对象相连。

```
object DetourDriver extends CompletableApp(5) {
```

```
...
```

```
val orderProcessor = system.actorOf(
  Props(new OrderProcessor()),
  "orderProcessor")
```

```
val debugger = system.actorOf(
  Props(new MessageDebugger(orderProcessor)),
  "debugger")
```

```
val tester = system.actorOf(
  Props(new MessageTester(debugger)),
  "tester")
```

```
val validator = system.actorOf(
```

器改道



```

    Props(new MessageValidator(tester)),
    "validator")

val logger = system.actorOf(
    Props(new MessageLogger(validator)),
    "logger")

val orderProcessorDetour = logger

orderProcessorDetour ! ProcessOrder(order)

awaitCompletion()

println("DetourDriver: is completed.")
}

```

这个改道器使 ProcessOrder 消息（代表处理订单命令）传输，经过 MessageLogger 对象（代表登录器）、MessageValidator 对象（代表验证器）、MessageTester 对象（代表测试器）、MessageDebugger 对象（代表调试器），最后将消息传输给 OrderProcessor 对象（代表订单处理器）。如果你想关闭这个改道器，可使用下面的方式：

```

object DetourDriver extends CompletableApp(5) {
    ...
    val orderProcessor = system.actorOf(
        Props(new OrderProcessor()),
        "orderProcessor")
    ...

    val orderProcessorDetour = orderProcessor

    orderProcessorDetour ! ProcessOrder(order)

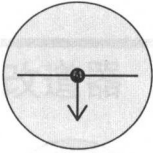
    awaitCompletion()

    println("DetourDriver: is completed.")
}

```

这会使 ProcessOrder 消息被直接发送给 OrderProcessor 对象。

窃听器



使用窃听器可以检查在 Actor 对象之间传输的消息。窃听器和改道器有很多相同点。它们之间的主要差异之一是，窃听器只能检查消息的内容，而改道器既能检查消息的内容又能修改消息的内容。

它们之间的另一个差异是，窃听器更像单个的过滤组件，而改道器更像组合到一起的多个过滤处理步骤。在使用窃听器时，消息既会被传送到它的最终目的地，也会在窃听器本身不执行检查处理步骤的情况下，被传输给执行检查操作的 Actor 组件。

下面示例中的窃听器被用于执行消息登录操作：

```
object WireTapDriver extends CompletableApp(2) {
  ...
  val orderProcessor = system.actorOf(
    Props(new OrderProcessor()),
    "orderProcessor")

  val logger = system.actorOf(
    Props(new MessageLogger(orderProcessor)),
    "logger")

  val orderProcessorWireTap = logger

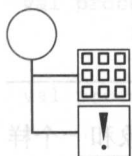
  orderProcessorWireTap ! ProcessOrder(order)

  awaitCompletion()

  println("WireTapDriver: is completed.")
}
```

MessageLogger 对象会登录所有发送给 OrderProcessor 对象的消息，然后将这些消息转发给 OrderProcessor 对象。在这类情况中，MessageLogger 对象既是一个窃听器，又是一个用于登录消息的过滤组件。

消息元数据/历史记录



将消息元数据添加到消息中，可以将多个用户、消息的发送者和接收者 Actor 与通过消息执行的操作关联起来。*Enterprise Integration Patterns* 一书中介绍过，可以将消息历史记录视为一种调试和分析工具。我将这种模式进一步总结为消息元数据，因为你选择的与消息关联的元数据可以通过任何方式被使用。你甚至能够使用元数据解决并发业务操作之间的冲突，因而可将元数据视为一种故障恢复手段。元数据很有用，其用途很广。

下面是将元数据添加到消息中的一种通用方式：

```
object Metadata {
  def apply() = new Metadata()

  case class Who(name: String)
  case class What(happened: String)
  case class Where(actorType: String, actorName: String)
  case class Why(explanation: String)

  case class Entry(who: Who, what: What, where: Where,
    when: Date, why: Why) {
    def this(who: Who, what: What, where: Where, why: Why) =
      this(who, what, where, new Date(), why)

    def this(who: String, what: String, actorType: String,
      actorName: String, why: String) =
      this(Who(who), What(what), Where(actorType, actorName),
        new Date(), Why(why))

    def asMetadata = (new Metadata()).including(this)
  }
}

import Metadata._

case class Metadata(entries: List[Entry]) {
  def this() = this(List.empty[Entry])
  def this(entry: Entry) = this(List[Entry](entry))
}
```



```
def including(entry: Entry): Metadata = {
  Metadata(entries :+ entry)
}
}
```

这个 Metadata 类（代表添加元数据的工具）自带了一个伴生对象和一个样本类。该 Metadata 容器由 List 类型的 Entry 集合构成。每个 Entry 集合都含有对 who、what、where、when 和 why 等单词的完整描述。要创建新的空白的 Metadata 实例，可以定义一个 Entry 集合，然后使用 asMetadata() 方法将 Entry 集合设置为元数据。

```
val entry = Entry(
  Who("user"),
  What("Did something"),
  Where(this.getClass.getSimpleName, "component1"),
  new Date(),
  Why("Because..."))
```

```
val metadata = entry.asMetadata
```

下面创建用于携带元数据的消息类型：

```
case class SomeMessage(
  payload: String,
  metadata: Metadata = new Metadata()) {

  def including(entry: Entry): SomeMessage = {
    SomeMessage(payload, metadata.including(entry))
  }
}
```

消息类型 SomeMessage 拥有简单的 String 类型的 payload 字段和一个 Metadata 实例，该 Metadata 实例默认情况下会含有空的 List 集合。如果你命令一个 SomeMessage 实例复制自己，但要在其 Metadata 实例中包含新的 Entry 集合，就会得到拥有相同 payload 值的新 SomeMessage 实例，但这个新 SomeMessage 实例会含有新的带有额外 Entry 集合的 Metadata 对象。

下面展示了在 Actor 对象之间使用添加了元数据的消息的方式。先从 Message-MetadataDriver 对象（应用程序对象）开始。

```
object MessageMetadataDriver extends CompletableApp(3) {
```

```

import Metadata._

val processor3 = system.actorOf(
  Props(new Processor(None)),
  "processor3")
val processor2 = system.actorOf(
  Props(new Processor(Some(processor3))),
  "processor2")
val processor1 = system.actorOf(
  Props(new Processor(Some(processor2))),
  "processor1")

val entry = Entry(
  Who("driver"),
  What("Started"),
  Where(this.getClass.getSimpleName, "driver"),
  new Date(),
  Why("Running processors"))

processor1 ! SomeMessage("Data...", entry.asMetadata)

awaitCompletion
}

```

该应用程序(MessageMetadataDriver)创建了3个Actor实例(Processor), 每个Processor对象都被赋予了具有唯一性的名称。这些Processor对象被连接到一起, 使processor1对象能够向processor2发送消息, processor2对象能够向processor3对象发送消息。然后, 该程序使用单词who、what、where、when和why, 创建了Entry集合。之后, Entry集合被转换为构成SomeMessage实例的元数据。然后, 这个SomeMessage消息被发送给processor1对象。

下面是Actor类Processor的代码:

```

class Processor(next: Option[ActorRef]) extends Actor {
  import Metadata._

  val random = new Random()

  def receive = {
    case message: SomeMessage =>
      report(message)

    val nextMessage = message.including(entry)

    if (next.isDefined) {

```

```

        next.get ! nextMessage
    } else {
        report(nextMessage, "complete")
    }

    MessageMetadataDriver.completedStep
}

def because = s"Because: ${random.nextInt(10)}"

def entry =
    Entry(Who(user),
        What(wasProcessed),
        Where(this.getClass.getSimpleName,
            self.path.name),
        new Date(),
        Why(because))

def report(message: SomeMessage,
    heading: String = "received") =
    println(s"${self.path.name} $heading: $message")

def user = s"user${random.nextInt(100)}"

def wasProcessed = s"Processed: ${random.nextInt(5)}"
}

```

当某个 Processor 对象收到 SomeMessage 消息后，该对象会报告收到该消息的情况，然后使用该 Processor 对象处理消息中附加的 Entry 集合的结果，创建一个新的 SomeMessage 实例。然后，如果该 Processor 对象被赋予下一个 Processor 对象，该 Processor 对象就会将新的 SomeMessage 消息发送给下一个 Processor 对象。然而，如果 processor3 对象收到了新的 SomeMessage 消息，它就会报告所有处理步骤都完成了。

运行该示例程序可得到下列结果（为适应书页篇幅调整了显示格式）：

```

processor1 received: SomeMessage(Data...,
Metadata(List(Entry(Who(driver),What(Started),
Where(MessageMetadataDriver$,driver),Wed Apr 30 15:57:41 MDT
2014,Why(Running processors))))))
processor2 received: SomeMessage(Data...,
Metadata(List(Entry(Who(driver),What(Started),
Where(MessageMetadataDriver$,driver),Wed Apr 30 15:57:41 MDT
2014,Why(Running processors)), Entry(Who(user50),
What(Processed: 4),Where(Processor,processor1),Wed Apr 30

```

```

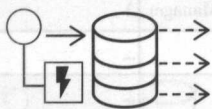
15:57:41 MDT 2014,Why(Because: 0))))
processor3 received: SomeMessage(Data...,
Metadata(List(Entry(Who(driver),What(Started),
Where(MessageMetadataDriver$,driver),Wed Apr 30 15:57:41 MDT
2014,Why(Running processors)), Entry(Who(user50),
What(Processed: 4),Where(Processor,processor1),
Wed Apr 30 15:57:41 MDT 2014,Why(Because: 0)),
Entry(Who(user80),What(Processed: 4),Where(Processor,
processor2),Wed Apr 30 15:57:41 MDT 2014,Why(Because: 1))))
processor3 complete: SomeMessage(Data...,
Metadata(List(Entry(Who(driver),What(Started),
Where(MessageMetadataDriver$,driver),Wed Apr 30 15:57:41
MDT 2014,Why(Running processors)), Entry(Who(user50),
What(Processed: 4),Where(Processor,processor1),Wed Apr
30 15:57:41 MDT 2014,Why(Because: 0)), Entry(Who(user80),
What(Processed: 4),Where(Processor,processor2),Wed Apr 30
15:57:41 MDT 2014,Why(Because: 1)), Entry(Who(user71),
What(Processed: 2),Where(Processor,processor3),Wed Apr
30 15:57:41 MDT 2014,Why(Because: 3))))

```

如该输出结果所示,新的 Entry 集合被包含在每个处理步骤生成的新消息中。

虽然此处没有详细介绍,但使用消息元数据可以为必须严格按照次序跨网络复制的消息序列 [Bolt-on Causal Consistency], 实现因果一致性 [AMC-Causal Consistency]。

消息日志/存储器



使用消息日志可以支持确保送达机制和持久订阅者。被写入日志的消息,可以代表某个 Actor 对象在其生命周期的特定阶段切换状态的情况,而且还可以被用于向接收者 Actor 对象执行后续发送操作。

可以将消息存储器用作保存已发送消息的永久存储位置。消息存储器的要点是,使用它可以更轻松地在将来分析以前发送和接收的消息。尽管可以通过为消息存储器设计强架构,使查询操作变得容易执行,但大多数情况中最好使用弱架构,并仅处理原始形式的消息。在根据消息的对象属性检查消息时,可通过消息存储器和消息流重组已存储的消息。如果你确实想要使用(较)强架构,使用关系数据库或者支持 JavaScript 对象表示法(JSON)或可扩展标记语言(XML)查询操作的数据库,可以做到这一点。例如,PostgreSQL 数据库支持保存 JSON 格

式数据的功能，和根据 JSON 属性执行的查询操作。

实际上，可以使用相同的机制实现消息日志和消息存储器。Akka Persistence 插件可以在消息被发送后，清理消息日志中的任何消息。然而，如果你将来还要使用这些消息，可以将它们保存在消息日志中，这样消息日志就成为消息存储器了。还可以使用 Akka Persistence 插件中的 View 工具创建独立的消息存储器，如图 10.1 所示。

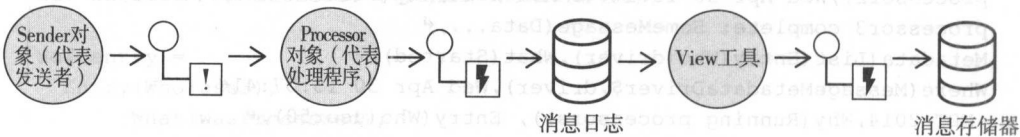


图 10.1 一条事件消息先被写入消息日志，之后又被保存到消息存储器中。

在这类情况中，Sender 对象向 Processor 对象发送了一条命令消息。一旦 Processor 对象完成了处理过程，就会生成一条事件消息。在该事件消息被发送给 View 工具前，该消息就会被写入消息日志中。一旦 View 工具收到了这条事件消息，就会将之保存到独立的消息存储器中。尽管这看起来不像 Akka Persistence View 工具的恰当用法，但事实并非如此。并非只能使用 View 工具生成命令查询职责分离（CQRS）型的用户接口视图，或生成某类报告。这些仅是 View 工具与窃听器类似的部分功能。

通常，消息日志和消息存储器拥有如图 10.2 所示的逻辑格式。

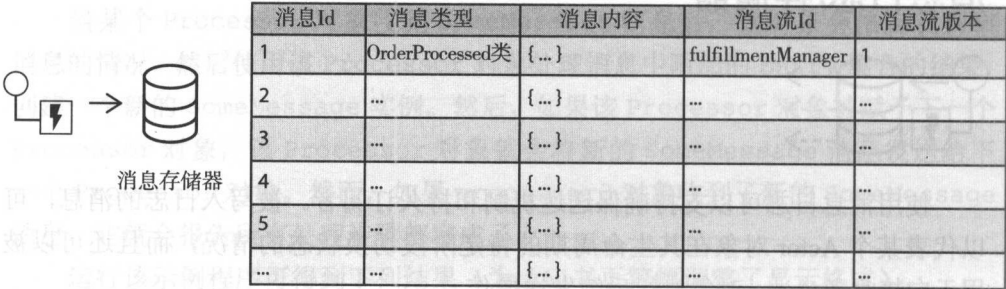


图 10.2 消息日志的可选和必选部分。

因为 Akka Persistence 插件允许自定义消息日志和消息存储器的结构，消息存储器产品可能会拥有图 10.2 所示的典型结构。下面介绍该表中各列的含义。

- **消息 Id**：每条消息都拥有唯一的身份，该身份是按照顺序生成的。这使所有在消息日志 / 存储器中被持久化的消息都能够被识别出其身份和次序。然而，这也代表了一个潜在的瓶颈，因为以递增方式分配消息 ID 的操作

不是可伸缩的操作。因此，这是一个可选的列（或字段），在消息日志中较少被实现。

- **消息类型**：消息存储器必须含有消息的类型，这样才能重组被序列化的消息。
- **消息内容**：这是被序列化的消息数据，这些数据可能是二进制数，也可能是文本（JSON 或 XML 格式的）。
- **消息流 Id**：这是持久化消息的接收者 Actor 对象的唯一身份（如 Processor 对象的 processorId 标识符）。换言之，每个 Actor 对象都拥有本身的消息流，这就是该 Actor 对象的主要标识。
- **消息流版本**：这是一种索引（从 1 开始，每增加一条消息该值都会加 1），用于描述 Actor 对象接收消息的次序。将消息流 Id 和消息流版本组合起来，可以描述指定消息的接收次序。消息流 Id：1 至消息流 Id：N，描述了一个 Actor 对象的整条消息流。

如果某个 Actor 对象（如 fulfillmentManager）收到了 100 条消息，那么它的消息流就是从消息流 Id：1 至消息流 Id：100，并且会根据消息流版本以升序被存储。因此，如果名为 fulfillmentManager 的持久化 Actor 对象，需要通过已存储的消息重组它的状态，通过按次序重新处理从消息流 Id：1 至消息流 Id：100 的消息，可以做到这一点。

默认的 Akka Persistence 消息日志实际上没有使用前面介绍的结构，而是使用 LevelDB 数据库实现了图 10.3 所示的结构。

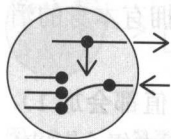


图 10.3 默认的 Akka Persistence 消息日志使用 LevelDB 数据库实现的逻辑结构。

这种结构能够帮助存储消息的操作获得极快的速度。每个持久化 Actor 对象都拥有本身的消息流版本（sequenceNr），每持久化一条事件消息后，sequenceNr 的值都会被增加。因为该结构中没有独立的顺序消息 Id，所以也不会执行插入操作时出现瓶颈。

该结构的缺点是所有在消息日志 / 存储器中被持久化的消息，都没有具有唯一性的身份和次序标识符。这意味着不能像从消息 Id : 1 至消息 Id : N 那样，从消息日志 / 存储器中简单地从头至尾读取所有消息。但使用因果一致性技巧 [AMC-Causal Consistency]，也可以实现相同的效果。

智能代理



在设计窃听器时，如果需要知道使用原始发送者的返回地址处理整个请求—回复过程的方式，可使用智能代理。在某些情况中，窃听器只需分析收到的请求消息，然后它只需将请求消息发送给原定的接收者 Actor 对象。但是在另一些情况中，窃听器还需要负责接收和分析回复的消息，然后将回复的消息发送给原始发送者。

图 10.4 展示了一个智能代理示例，和整个处理过程。

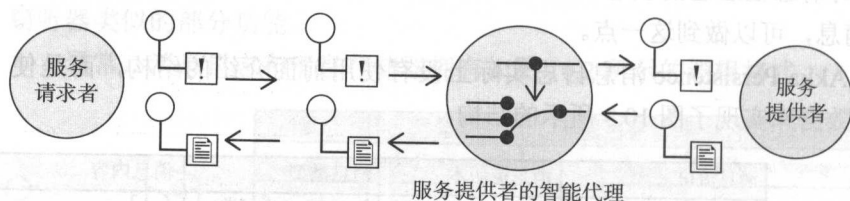


图 10.4 智能代理常用的处理过程。

让我们先观察下 Actor 对象 ServiceProvider (代表服务提供者)，该对象最终会接收所有请求消息。

```
class ServiceProvider extends Actor {
  def receive = {
    case one: ServiceRequestOne =>
      sender ! ServiceReplyOne(one.requestId)
    case two: ServiceRequestTwo =>
      sender ! ServiceReplyTwo(two.requestId)
    case three: ServiceRequestThree =>
      sender ! ServiceReplyThree(three.requestId)
  }
}
```


这个 `ServiceProvider` 对象没有执行过多操作，但你能够预料为了满足收到的请求消息，需要做大量的工作。然而，此处的要点是，当 `ServiceProvider` 对象收到请求消息时，就必须向请求者发送回复消息。创建好回复消息后，`ServiceProvider` 对象必须为回复消息赋予请求消息的 `requestId` 值（代表消息 ID），这使代理能够将具体的回复消息和相应的请求消息关联起来。

下面是服务必须支持的请求消息和回复消息，它们使用了 Scala 语言中的特征和统一访问原则²[Meyer-OOSC]，创建了常见的消息类型 `ServiceRequest`（代表请求消息）和 `ServiceResponse`（代表回复消息）。

```

trait ServiceRequest {
  def requestId: String
}

case class ServiceRequestOne(requestId: String) extends ServiceRequest
case class ServiceRequestTwo(requestId: String) extends ServiceRequest
case class ServiceRequestThree(requestId: String) extends ServiceRequest

trait ServiceReply {
  def replyId: String
}

case class ServiceReplyOne(replyId: String) extends ServiceReply
case class ServiceReplyTwo(replyId: String) extends ServiceReply
case class ServiceReplyThree(replyId: String) extends ServiceReply

```

通过发送 3 个消息对象之一可以执行请求操作，这些消息包括：`ServiceRequestOne`、`ServiceRequestTwo` 和 `ServiceRequestThree`。在 `ServiceProvider` 对象执行回复操作时，它必须发送 3 条消息之一：`ServiceReplyOne`、`ServiceReplyTwo` 和 `ServiceReplyThree`。常见的消息处理模式是，必须使它们包含 `requestId` 或 `replyId` 字段（分别代表请求消息和回复消息的 ID）。而且你会发现，这是正确实现智能代理的关键。

² “模块应该通过统一的表示方式提供所有服务，不论这些服务是通过存储设备提供的还是通过计算提供的。”在这类情况中，你无法分辨出 `requestId` 和 `replyId` 是属性还是函数，也无法确定它们被声明和定义的位置。

下面让我们看看向 ServiceProvider 对象请求获取服务的 Actor 对象 (ServiceRequester):

```
case class RequestService(service: ServiceRequest)

class ServiceRequester(serviceProvider: ActorRef) extends Actor {
  def receive = {
    case request: RequestService =>
      println(s"ServiceRequester: ${self.path.name}: $request")
      serviceProvider ! request.service
      SmartProxyDriver.completedStep
    case reply: Any =>
      println(s"ServiceRequester: ${self.path.name}: $reply")
      SmartProxyDriver.completedStep
  }
}
```

ServiceRequester 对象会接收 RequestService 消息, 该消息会命令它向 ServiceProvider 对象请求获取服务。在本例中, RequestService 消息含有具体的 ServiceRequest 消息, 这条 ServiceRequest 消息会被发送给 ServiceProvider 对象, 以便请求获取服务。

前面介绍的代码中缺少的 Actor 对象是 ServiceProviderProxy, 下面是这个智能代理的实现代码:

```
class ServiceProviderProxy(serviceProvider: ActorRef) extends Actor {
  val requesters = scala.collection.mutable.Map[String, ActorRef]()

  def receive = {
    case request: ServiceRequest =>
      requesters(request.requestId) = sender
      serviceProvider ! request
      analyzeRequest(request)
    case reply: ServiceReply =>
      val sender = requesters.remove(reply.replyId)
      if (sender.isDefined) {
        analyzeReply(reply)
        sender.get ! reply
      }
  }

  def analyzeReply(reply: ServiceReply) = {
    println(s"Reply analyzed: $reply")
  }
}
```

```
def analyzeRequest(request: ServiceRequest) = {
  println(s"Request analyzed: $request")
}
}
```

ServiceProviderProxy 对象的一项工作是，接收所有 ServiceRequest 消息，分析它们然后将它们发送给 ServiceProvider 对象。ServiceProviderProxy 对象的另一项工作是，接收所有 ServiceReply 消息，分析它们然后将它们发送给 ServiceRequester 对象（即原始请求者）。要使这种处理过程变得完整（使请求消息与回复消息一一对应），智能代理必须创建相关标识符，并将之与 ServiceRequester 对象的 ActorRef 引用关联起来。该相关标识符会被添加到智能代理收到的 ServiceRequest 消息中。在这种情况下，该相关标识符能够拥有唯一性，因为通过在应用程序中进行设计可以轻松做到这一点。当在应用程序中需要通过设计使智能代理生成具有唯一性的相关标识符时，可使用 java.util.UUID 类。

```
val requestId = UUID.randomUUID.toString
val request = ServiceRequestOne(requestId)
```

通过这种方式获取具有唯一性的标识符时，一个明显的问题是，智能代理必须知道通过将相关标识符添加到它收到的每条请求消息中，使这些消息拥有唯一性的方式。否则，智能代理就需要将所有 ServiceRequest 消息都添加到一个封装器中。

```
class ServiceProviderProxy(serviceProvider: ActorRef) extends Actor {
  val requesters = scala.collection.mutable.Map[String, ActorRef]()

  def receive = {
    case request: ServiceRequest =>
      requesters(request.requesterIdentity) = sender
      val requestId = UUID.randomUUID.toString
      val envelope = ServiceRequestEnvelope(requestId, request)
      serviceProvider ! envelope
      analyzeRequest(request)
    ...
  }
}
```

然而，这样做意味着 ServiceProvider 对象还要担负解除请求消息封装的额外工作。最糟糕的情况是，必须使 ServiceProvider 对象了解智能代理和窃

听器的存在。最理想的情况是服务请求者是可以信赖的，能够提供可用作相关标识符的具有唯一性的标识符。如果由于某种原因无法获得最理想情况，你就不得不选择较困难的处理方式。

下面是该示例程序 (SmartProxyDriver) 的代码：

```
object SmartProxyDriver extends CompletableApp(6) {
  val serviceProvider = system.actorOf(
    Props[ServiceProvider],
    "serviceProvider")
  val proxy = system.actorOf(
    Props(new ServiceProviderProxy(serviceProvider)),
    "proxy")
  val requester1 = system.actorOf(
    Props(new ServiceRequester(proxy)),
    "requester1")
  val requester2 = system.actorOf(
    Props(new ServiceRequester(proxy)),
    "requester2")
  val requester3 = system.actorOf(
    Props(new ServiceRequester(proxy)),
    "requester3")
  requester1 ! RequestService(ServiceRequestOne("1"))
  requester2 ! RequestService(ServiceRequestTwo("2"))
  requester3 ! RequestService(ServiceRequestThree("3"))
  awaitCompletion
}
```

该程序先创建了关键的 ServiceProvider 对象，然后创建了 ServiceProviderProxy 对象（代表智能代理）。该智能代理被赋予了 ServiceProvider 对象的引用。之后，该程序创建了 3 个 ServiceRequester 实例，每个实例都被赋予了 ServiceProviderProxy 对象的引用。最后，每个请求者 Actor 对象都被命令向指定的服务器发出请求。下面是上述处理过程的输出结果：

```
ServiceRequester: requester1:␣
  RequestService(ServiceRequestOne(1))
ServiceRequester: requester2:␣
  RequestService(ServiceRequestTwo(2))
ServiceRequester: requester3:␣
```

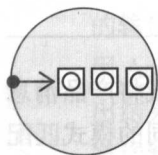
```

RequestService(ServiceRequestThree(3))
Request analyzed: ServiceRequestOne(1)
Request analyzed: ServiceRequestTwo(2)
Request analyzed: ServiceRequestThree(3)
Reply analyzed: ServiceReplyOne(1)
ServiceRequester: requester1: ServiceReplyOne(1)
Reply analyzed: ServiceReplyTwo(2)
ServiceRequester: requester2: ServiceReplyTwo(2)
Reply analyzed: ServiceReplyThree(3)
ServiceRequester: requester3: ServiceReplyThree(3)

```

通过这种方式将智能代理用作窃听器，可以对 Akka 控制总线没有提供的消息执行特殊的分析任务（专门针对应用程序的）。

测试消息



当需要查明消息是否已经被妥善接收时，可向 Actor 对象发送测试消息。妥善接收消息不仅是指消息被送达，还包括由消息运载的所有数据都被送达。

因为在默认情况下，Akka 框架为每个 Actor 对象都提供了一个无界的消息缓存 (UnboundedMailbox)，所以当许多 Actor 对象收到许多消息时，就可能会耗尽 JVM 的内存。如果使用其他类型的消息缓存（如 BoundedMailbox），就有可能在消息缓存被装满时丢失消息。当向远程 Actor 对象发送消息时，网络可能会出故障或不稳定，因而导致彻底丢失消息或使消息含有错乱的数据。因此，向系统中的各种 Actor 对象发送测试消息，就会使你得到各种好处。可以使用两种基本方式创建测试消息。可以使用常规的合法消息装载测试数据，需要为这类消息添加格式标识符，以表明其运载的数据仅用于测试。换言之，不应像处理真正的产品级消息那样处理测试消息。使用这种创建方式时会遇到一点麻烦，因为你必须创建拥有正确格式的消息，这意味着必须生成具有真实效果的测试数据。有时找出测试数据非常困难，或者说找出能够使系统以正常模式运行的测试数据非常困难。也许最简单的方式是，将真正的产品级消息复制到带有格式标识符的测试消息中。一旦你确定了创建测试消息的适当方式，那么就应该在接收者 Actor 对象中放置合适的分支逻辑，以避免测试数据被当作产品数据处理，以及在测试

代码中丢失产品数据。应谨慎，不应想当然地处理这个步骤。

```

trait Testable {
  def isTest(): Boolean
}

case class ProcessOrder(..., forTest: Boolean) extends Testable {
  def this(...) = { this(..., false) } // default construction
  override def isTest(): Boolean = { forTest }
  ...
}

class OrderProcessor extends Actor {
  case processOrder: ProcessOrder if (processOrder.isTest) =>
    // 在此处测试这条消息
  case processOrder: ProcessOrder =>
    // 在此处处理产品信息
  ...
}

```

第二种创建测试消息的方式是设计含有测试数据的独立消息，或将产品消息的副本用作测试消息。该方式的效果很好，因为你能够使用完全不同的模式匹配功能测试消息。

```

case class TestMessage(actualMessage: Any)

class OrderProcessor extends Actor {
  case processOrder: ProcessOrder =>
    // 在此处处理产品信息
  case testMessage: TestMessage =>
    // 在此处测试这条消息
  ...
}

```

这也许是支持测试消息的最安全方式，因为这消除了下列潜在问题：

- 将产品信息（如 ProcessOrder）标记为测试消息。
- 将测试消息（如 ProcessOrder）标记为产品信息。
- Actor 对象的模式匹配语句没有设置好分支。
- 消息格式改变了，而接收者 Actor 对象错误地以旧方式处理新消息。

Enterprise Integration Patterns 一书介绍了许多可用于为测试消息提供彻底支持的组件。

- **测试数据生成器**：该 Actor 对象会使用上述两种方式中的一种创建测试消息。这意味着你可以通过该用户接口根据 Actor 对象支持的消息类型，生成特定类型的测试消息。
- **测试消息注入器**：该 Actor 对象实际上用于对目标 Actor 对象执行发送消息的操作。这意味着你可以通过该用户接口以动态方式使用 Actor 系统查看 Actor 对象，然后从中选择测试消息注入器支持的消息类型并发送这些消息。使用这类用户接口还可以设置发送消息的总数、发送消息的频率等。
- **测试消息分离器（或测试结果处理器）**：如果你在测试消息中包含了由目标 Actor 对象处理的分析函数，那么这些目标 Actor 对象就应该将它们的输出结果发送给测试消息分离器。测试消息分离器本身是 Actor 对象，用于接收测试结果并使用与系统相匹配的方式处理它们。该组件被命名为分离器的原因是，在 *Enterprise Integration Patterns* 一书中介绍的某种环境中，该组件会起到基于内容的路由器的作用，在一个消息通道中传输程序的输出结果，在另一个消息通道中测试程序的输出结果。然而，因为在使用 Actor 模型时消息通道就是 Actor 对象的消息缓存，所以程序的所有输出结果都会自然而然地被发送给产品级 Actor 对象，而测试输出结果会被发送给测试消息分离器。在这类情况中，测试消息分离器就是一个测试结果处理器。
- **测试数据验证器**：测试数据验证器是一种分析函数，用于接收测试结果处理器发出的处理结果并验证这些测试数据。测试数据验证器既会验证收到的测试消息，也会验证向外发出的测试结果消息。测试结果处理器会起到多个测试数据验证器的监督者的作用，这些测试数据验证器代表了多个分析步骤，会接收需要验证的消息。

请参阅 *Enterprise Integration Patterns* 一书，详细了解这些组件的用法，并做好使用基于 Actor 的模式实现它们的思想准备。

通道净化器



在使 Actor 对象支持确保送达机制，而且在进行测试和正常运行程序前，必

须从消息存储器中删除一些消息的情况中，可使用通道净化器。某些没有使用确保送达机制的 Actor 对象，可能也必须清除消息缓存中保存的消息。而且活动系统中的某些消息，有时需要被丢弃而不被处理。

没有使用确保送达机制的 Actor 系统会自动清理 Actor 对象的消息缓存，Actor 对象停止运行时所有消息缓存都会被清空。然而，使用了确保送达机制的 Actor 系统和支持至少发送一次机制的 Actor 对象，都需要支持清理消息缓存的功能。

如果使用了确保送达机制和消息存储器，那么创建实现通道净化器模式的 Actor 对象和其他组件就会比较简单。通道净化器会直接从消息存储器中删除部分或全部消息。可以通过识别发送者 Actor 对象、原定接收者 Actor 对象或二者兼有的身份，使用通道净化器过滤掉需要删除的消息。

另一方面，如果你想仅支持普通的至多发送一次消息缓存，删除指定 Actor 对象消息缓存中的消息就会有点困难。这是因为，默认为 Actor 对象分配的 UnboundedMailbox（默认的消息缓存类型），是一种先进先出（FIFO）的队列，而且 Actor 对象的接收代码块被设置为，同一时刻只能按照消息被保存到消息缓存中的次序接收每一条消息。因此，如果你向这类 Actor 对象发送删除其消息缓存中的消息的命令消息，除非该 Actor 对象处理完此前收到的所有消息，否则该 Actor 对象不会处理你发送的删除其消息缓存中的消息的命令消息。在这种情况下，也许删除 Actor 对象消息缓存中的消息的最简单方式是，停止该 Actor 对象然后重启它。

```
system.stop(actorRef)
```

```
// 或者 ...
```

```
context.stop(childRef)
```

停止 Actor 对象会导致该 Actor 对象结束当前消息的处理过程；将其消息缓存中保存的所有消息都清空，而且该 Actor 对象也会停止运行。注意，使用 PoisonPill 消息无法像使用 stop() 方法那样立刻使 Actor 对象停止运行。PoisonPill 消息会以正常的次序进入 Actor 对象的消息缓存（默认情况下是一种 FIFO 队列），并按次序被 Actor 对象处理。因此，在使用 FIFO 消息缓存时，排在 PoisonPill 消息前面的消息会先被 Actor 对象处理。Kill 消息也会以 PoisonPill 消息被处理的方式被处理，但是当 Kill 消息被 Actor 对象收到时，Kill 消息会使 Actor 对象抛出 ActorKilledException 异常。当收到 Kill 消息的 Actor 对象

的监督者收到这个异常时，会做出重启该 Actor 对象的决定。此处的要点是，如果你想使用预制功能立刻清空 Actor 对象的消息缓存，就应使用 `stop()` 方法。

使 Actor 对象停止并重启的另一种方式是，将带优先权机制的队列（如 `PriorityBlockingQueue`）用作消息缓存，从而使你能够向 Actor 对象发送拥有较高优先权的清空消息缓存的消息。当 Actor 对象收到清空消息的命令后，该 Actor 对象会变成一个无底洞，立刻清空其消息缓存中的所有消息并抛弃收到的其他消息。

```
class OrderProcessor extends Actor {
  ...
  def normal: Receive = {
    case processOrder: ProcessOrder =>
      ...
    case purge: PurgeNow =>
      context.become(purger)
  }

  def purger: Receive = {
    case noPurge: StopPurge =>
      context.become(normal)
    case ignore: Any =>
  }

  def receive = normal
}
```

当然，你还需要向该 Actor 对象发送一条优先权较低的消息，命令该 Actor 对象停止清空其消息缓存和丢弃收到的消息，以便使之能够重新正常地处理消息。否则，该 Actor 对象（如本例中的 `OrderProcessor`）就永远无法重新变成正常的订单处理器。

小结

本章介绍了可在开发、测试和 Actor 系统产品环境中使用的各种工具。这些工具中最突出的两种是消息元数据和消息日志，大多数基于 Actor 的应用程序都会用到这两种工具。在工作中，有时你还会用到改道器、窃听器、智能代理、测试消息和通道净化器等优秀工具。

附录A

.NET平台上的Akka工具集: Dotsero

本附录介绍 Dotsero[Dotsero Toolkit]，它是一个易用且功能强大的工具集，用于在 .NET 平台上使用 Actor 模型开发程序。本附录还会介绍通过 C# 编程语言使用 Actor 对象的基础知识。对 Java 或 Scala 语言了解较少的程序员，会更容易接受这部分内容。通过使用请求—回复模式试验 Actor 对象的基础消息传输模式，可以揭开 Actor 对象的神秘面纱。

Dotsero的Actor系统

Dotsero 支持 Akka 工具集的大多数基础功能，而且使用这些功能的方式也很相似。图 A.1 展示了 Dotsero 支持的主要功能。

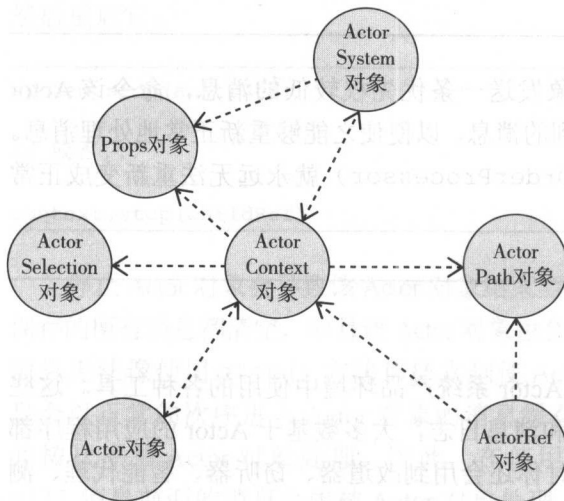


图 A.1 Dotsero 支持的 Actor 系统中的基础元素。

如图 A.1 所示，Dotsero 支持 Akka 框架中的主要概念，其中包括：ActorSystem、

Props、Actor、ActorRef、ActorContext、ActorSelection 和 ActorPath。这意味着 Dotsero 会支持上述对象中含有的方法提供的大量功能。因为 Dotsero 也通过与 Scala 语言的 Akka 框架使用这些对象的方式相同的方式使用这些对象，所以我不会详细介绍通过 Dotsero 使用这些对象的方式。但是，为了确定在 Dotsero 中也是通过相同方式使用这些对象的，我列出一些简单的代码。

下面的代码展示了创建和关闭 ActorSystem 对象的方式：

```
ActorSystem system =
    ActorSystem.Create("ActorSystemTests");
Assert.AreEqual("ActorSystemTests", system.Name);
system.Shutdown();
```

这段示例代码断言了 ActorSystem 对象的名称，就是在创建该 ActorSystem 对象时使用的名称。使用下面的代码还可以获取该 ActorSystem 对象启动的时间，或该 ActorSystem 对象持续正常运行的时间。

```
Assert.IsTrue(system.Uptime.Ticks > 100);
```

该 ActorSystem 对象在根节点以下有两条主要路径：用户守护对象（user）和系统守护对象（sys）。

```
Assert.AreEqual("/user", system.Context.Path.Value);
Assert.IsFalse(system.Context.Path.IsRoot());
Assert.AreEqual("/sys", system.SystemContext.Path.Value);
Assert.IsFalse(system.Context.Path.IsRoot());
Assert.AreEqual(ActorPath.RootName,
    system.Context.Parent.Path.Value);
Assert.IsTrue(system.Context.Parent.Path.IsRoot());
```

该 ActorSystem 对象还拥有一个伪 Actor 对象 DeadLetters，该对象用于支持死信通道。

```
Assert.AreEqual("deadLetters",
    system.DeadLetters.Path.Name);
```

如你尝试向某个不存在的 Actor 对象发送消息，那么这条消息实际上就会被发送给 DeadLetters 对象。

```
ActorSelection selection =
    system.ActorSelection("/user/NonExistingActor42");
selection.Tell("TESTING DEAD LETTERS");
```

上面的代码表明 Dotsero 还支持 ActorSelection 对象, 使用 ActorSelection 对象可以找到 user 守护者对象下方的 Actor 对象。还可以通过 ActorContext 对象使用 ActorSelection 对象 (在 Actor 对象的内部), 在 ActorSelection 路径指向的 Actor 对象存在的情况下, 这样做可以找到指定上下文中的 Actor 对象。下面的示例代码向 user 守护者对象下方的一个 Actor 对象层次结构中的多个 Actor 对象发送了多条消息 (以自顶向下的次序):

```
system.ActorSelection("/user/sel1").Tell("TEST");
system.ActorSelection("/user/sel1/sel2").Tell("TEST");
system.ActorSelection("/user/sel1/sel2/sel3").Tell("TEST");
```

第一条消息被发送给 sel1 对象, 该对象位于 user 守护者对象的下方, 紧邻 user 守护者对象。第二条消息被发送给 sel2 对象, 该对象是 sel1 对象的一个子对象。第三条消息被发送给 sel3 对象, 该对象是 sel2 对象的子对象。

如图 A.2 所示, Dotsero 还支持 Actor 对象的监督机制, 允许父 Actor 对象监督它的子 Actor 对象。

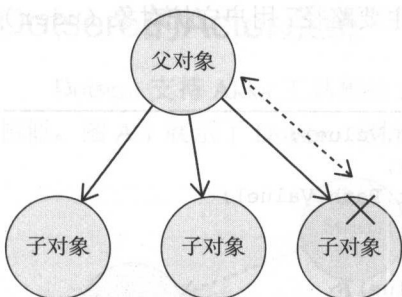


图 A.2 Dotsero 支持监督子对象的层次结构。

如果父 Actor 对象想要通过异常检测其子 Actor 对象的崩溃情况, 那么就必须注明监督策略。

```
public class EscalateSupervisorStrategy : SupervisorStrategy
{
    public EscalateSupervisorStrategy()
        : base(SupervisorStrategy.StrategyType.OneForOne)
    {
    }

    public override Directive Decide(Exception e)
    {
        return SupervisorStrategy.Directive.Escalate;
    }
}
```

```

    }
}

public class Level1 : Actor
{
    ...
}

public class Level2 : Actor
{
    public Level2()
    {
        SupervisorStrategy =
            new EscalateSupervisorStrategy();
    }
    ...
}

public class Level3 : Actor
{
    ...
    public void OnReceive(string message)
    {
        throw new InvalidOperationException(
            "TEST ESCALATE");
    }
}

```

在本例中，Actor 对象被划分为 3 个等级。Level1 是祖父对象、Level2 是子对象，而 Level3 是孙对象。Actor 对象 Level2 注册了特殊的监督策略，当它的子 Actor 对象 Level3 崩溃时，Level2 对象会使该崩溃问题升级，将该问题交给它的父对象 Level1 处理。因此，当孙对象 Level3 抛出 `InvalidOperationException` 异常时，最初会由它的父对象 Level2 处理该异常。然而，Level2 对象决定将该异常问题升级，将该异常交给它的父对象 Level1 处理。Level1 对象获得了监督权后，会使用默认监督策略（在其构造器没有设置其他监督策略的情况下），停止并重启它的子对象 Level2，进而导致 Level3 对象被重启。

通过C#和.NET使用Actor对象

在通过 Dotsero 提供基础的系统支持的情况下，让我们来观察 Actor 对象的运行方式。下面的示例使用了一个 Actor 对象 Manager 和一个 Actor 对象 Worker，

这两个 Actor 对象会执行基本的请求一回复式交互操作。让我们先看看 Actor 对象 Manager。

```

public class Manager : Actor
{
    private EventWaitHandle helloBackEvent;
    private ActorRef worker;

    public Manager(
        ActorRef worker,
        EventWaitHandle helloBackEvent)
        : base()
    {
        this.worker = worker;
        this.helloBackEvent = helloBackEvent;
    }

    public void OnReceive(string message)
    {
        switch (message)
        {
            case "start":
                worker.Tell("hello", Self);
                break;

            case "hello, back":
                helloBackEvent.Set();
                break;

            default:
                Assert.Fail("Invalid message type.");
                break;
        }
    }
}

```

Manager 类扩展了 Actor 基类, 因此 Manager 对象会实现大部分 Actor 对象都必须实现的基础协议。稍后会详细介绍 Actor 基类, 但此时你只需将注意力集中在 Manager 对象执行的操作上。

Manager 对象是通过几个参数创建的。第一个参数是指向另一个 Actor 对象 (即 Worker 对象) 的引用, Manager 对象会与该 Actor 对象协同工作。有趣的是, Manager 对象不必了解 Actor 对象 Worker 的具体数据类型, 它只需知道

它拥有指向 Worker 对象的 ActorRef 引用, 并可以通过该引用向 Worker 对象发送需要处理的工作。为 Manager 对象提供处理工作的 Actor 对象的 ActorRef 引用, 是 Manager 对象的直接客户端或父对象的职责。第二个参数是 EventWaitHandle, 它不是一个 Actor 对象, Manager 对象通过它与外部进行响应式通信, 以表明工作已经完成。Manager 对象使用 worker 字段保存 Worker 对象的 ActorRef 引用, 使用 helloBackEvent 字段保存 EventWaitHandle 参数。

创建并初始化了 Manager 对象后, 在收到使之工作的命令前, Manager 对象不会做任何事情。Manager 对象与普通对象之间既有相同点也有不同点。实际上, 你无法直接调用 Manager 对象中的某些行为型方法。但是, 可以通过向 Actor 对象 Manager 发送消息, 命令它处理工作。

```
Manager manager = new Manager(worker, doneEvent);
```

```
manager.Tell("start");
```

Tell() 方法不仅仅是一件被降低了耦合性的用于执行操作的死板工具。Tell() 方法背后的消息传输机制, 会使 Manager 对象具有响应性, 并使之能够在伸缩性较高的环境中工作, 还使程序员能够更轻松地推导出使用并发模式的时机。实际上, 如果消息传输机制没有这些优点, 许多 C# 和 Java 程序员会直接调用对象中的行为型方法, 从而使编程变得更简单。因此, 你可以命令 Manager 对象启动。让我们回头再看看 Manager 对象中的 OnReceive() 方法。该方法使 Actor 对象能够处理发送给该 Actor 对象的消息。你向 Actor 对象发送的消息的类型不仅限于 String, 还可以发送 int 或 double 类型的消息, 以及对象类型的消息。你需要做的仅是为每个消息类型创建重载的 OnReceive() 方法。不论你发送哪种数据类型的消息, 这些消息都应该是不可变的。这可以帮助你谨守 Actor 模型的不共享原则。

当 Manager 对象的 OnReceive() 方法被 start 消息调用时, Manager 对象会通过命令 Worker 对象显示单词 hello 对此做出回应。除了向 Worker 对象发送 hello 消息外, Manager 对象还会命令 Worker 对象通过解析它的 Self 引用, 了解 Manager 对象是该消息的发送者。虽然这显得怪异, 但你很快就会明白这一点为什么很重要。

```
...非创, 象扶 totaA 服家 (的球委带下) 器查的并想因使双会不游, 常照  
public void OnReceive(string message)  
{
```

```

switch (message)
{
    case "start":
        worker.Tell("hello", Self);
        break;
    ...
}
}

```

实际上, 现在就是将 Actor 对象 Worker 的实现代码展示出来的绝好时机:

```

public class Worker : Actor
{
    public Worker()
    {
    }

    public void OnReceive(string message)
    {
        Assert.AreEqual("hello", message);

        if (this.CanReply)
        {
            Sender.Tell("hello, back", Self);
        }
        else
        {
            Assert.Fail("Worker: no reply-to available...");
        }
    }
}

```

当 Worker 对象收到 hello 消息时, 它负责的全部工作是, 回复该消息的发送者 "hello, back" 消息。这就是在调用 Tell() 方法时, 在消息对象中添加 Self 参数的原因。这使接收者 Actor 对象能够回复发送者 Actor 对象。通过接收者 Actor 对象的 OnReceive() 方法的 Sender 属性, 可以获得 Self 参数 (发送者的 ActorRef 引用)。

这就像打乒乓球。虽然不完全相同, 但大部分基于 Actor 对象的编程方式与打乒乓球很像。基本的请求—回复模式被广泛使用。

通常, 你不会仅使用默认的构造器 (不带参数的) 实现 Actor 对象。除非在创建 Actor 对象时通过 Props 对象传递参数, 否则不必使 Actor 对象的构造器接

收参数。你可以将 Worker 对象的构造器与 Manager 对象的构造器（该构造器确实接收参数）做对比。

让我们再回过头来观察 Manager 对象，当该对象收到 "hello, back" 消息时，会通过发送 helloBackEvent 消息结束此次请求一回复处理过程。请思考下面的测试程序示例代码。

```
namespace DotseroTest
{
    using Dotsero.Actor;
    using Microsoft.VisualStudio.TestTools.UnitTesting;
    using System.Threading;

    [TestClass]
    public class ActorManagerWorkerTests
    {
        [TestMethod]
        public void ManagerUsesWorker()
        {
            ActorSystem system =
                ActorSystem.Create("ActorSystemTests");

            ActorRef worker =
                system.ActorOf(
                    typeof(Worker),
                    Props.None,
                    "worker");

            AutoResetEvent helloBackEvent =
                new AutoResetEvent(false);

            AutoResetEvent specificEvent =
                new AutoResetEvent(false);

            ActorRef manager =
                system.ActorOf(
                    typeof(Manager),
                    Props.With(worker, helloBackEvent),
                    "manager");

            manager.Tell("start");

            var helloBackDone =
```

```

        helloBackEvent.WaitOne(1000, false);

        system.Stop(worker);

        system.Stop(manager);

        system.Shutdown();

        Assert.IsTrue(helloBackDone);
    }
}

```

因为 ActorManagerWorkerTests 类不是一个 Actor 类, 所以测试方法 Manager-UsesWorker() 无法告诉 Actor 对象 Manager, ActorManagerWorkerTests 对象是消息发送者。换言之, 只是因为该测试程序无法将发送者的 ActorRef 引用作为第二个参数传送, 所以重载的 Tell() 方法才会接收 start 消息。测试程序可以接受这种处理方式, 而且这也是创建和使用 helloBackDone 断言, 表明请求一回复消息传输过程成功完成的原因。如果 Manager 对象没有收到 "hello, back" 消息, 那么 helloBackDone.Set() 方法就不会被调用, 因而测试方法的断言就会失败。

Dotsero实现

实际上 Dotsero 是使用开源的 Retlang 库实现的 [Retlang]。该库是由 Mark Rettig 编写的一些低等级抽象构成的, 通过这些抽象可以像使用 Erlang 语言一样, 在 .NET 平台上使用 C# 编写程序。Retlang 项目声称它实现了 Actor 模型, 但实际上它更像一种基于线程的低等级消息传输应用程序编程接口 (API)。因此, 事实上它不太像 Erlang 语言。我会在它的周围放置简单的 Actor 抽象, 因为单独使用 Retlang 库不是那么简单的, 至少不会像使用真正的 Actor 模型工具集, 如 Dotsero, 那样简单。

Retlang 库与 Akka 框架的异同点

对于 Retlang 库, Mark Rettig 提出的一个警告是, 它不像 Erlang 语言或 Akka 框架的 Actor 系统那样具有可伸缩性。Mark Rettig 提出, 为支持数千个 Actor 对象而创建数千个通道, 会降低性能。当然, 他的这个观点是在与直接调用对象方法做比较的情况下提出的, 而即使在消息使用少量间接调用方式的情况下,

直接调用对象中的方法总是具有最快的速度。但是，我已经介绍过，在提升应用程序的整体性能方面，Actor 系统有许多优点。通过并发和并行处理模式获得的性能提升，远超通过直接调用对象方法获得的性能提升，更不要说基于 Actor 的应用程序还拥有可伸缩性等更多的优点。

无论如何，如果你编写的系统至少含有数百或数千个 Actor 对象，就能够获得一定程度的多线程化性能和可伸缩性。你可能会将以 Actor 模型为基础的复合对象模型用作特殊工具，仅使用 Actor 模型处理应用程序中的关键热点问题。

然而，你会发现越多使用基于 Retlang 的 Actor 对象，获得的效果就越好（就像通过常见方式使用 Actor 模型一样）。这最终是由 Retlang 库的设计决定的。

另一方面，Akka 框架专门被设计为，通过可伸缩的方式在单台 Java 虚拟机中实现数百万个 Actor 对象。即使要处理数百万个 Actor 对象，你也能够确信应用程序的性能不会因此而降低。这一点是毋庸置疑的。

Dotsero 自带了两个对象：IChannel 和 IFiber，IChannel 实例是通过默认的 Channel 类创建的，而 IFiber 实例被实现为运载工具池。为了避免涉及 Retlang 实现的高深知识，我们此刻只需知道 IChannel 对象代表发送消息的方式，而 IFiber 对象代表传输消息的方式。我们需要了解的 Actor 基类的一个要点是，IFiber 是基于线程池的对象。当消息通过 IChannel 对象被发送时，在可获得线程的情况下，IFiber 对象会通过 .NET 系统中的 IThreadPool 接口，将消息传输给 Actor 对象。

当 ActorRef 对象调用了 Tell() 方法后，ActorRef 对象就会对 Delivery 对象执行入队操作。

```
public class ActorRef
{
    ...
    public void Tell(object message, ActorRef sender)
    {
        if (!Context.Terminated)
        {
            Context.Enqueue(new Delivery(message, sender));
        }
        else
        {
            Context

```

```

        .System
        .DeadLetters.Tell(message, sender);
    }
}

```

因此所有通过 Actor 类扩展的类（如 Manager 和 Worker），都能够接收任何类型的消息。你需要做的仅是为你编写的 Actor 对象支持的消息类型，创建不同的 OnReceive() 方法。下面我为 Manager 对象添加了新的 OnReceive() 方法，使之能够接收 SpecificMessage 类型的消息：

```

public class Manager : Actor
{
    ...
    public void OnReceive(SpecificMessage specificMessage)
    {
        ...
    }
}

public sealed class SpecificMessage
{
    public static SpecificMessage Create(string message)
    {
        return new SpecificMessage(message);
    }

    public SpecificMessage(string payload)
    {
        this.Payload = payload;
    }

    public string Payload { get; private set; }

    public override string ToString()
    {
        return "SpecificMessage: " + Payload;
    }
}

```

这样就使 Manager 对象能够接收 SpecificMessage 类型的消息了。注意，SpecificMessage 被声明为密封类，以确保使之拥有不可变性。这还确保了该类的 Payload 属性的读取器是公用的，而它的写入器是私有的。

当然,除非客户端(如某个 Actor 对象)使用两个重载 `Tell()` 方法其中之一,向你编写的 Actor 对象发送消息,否则你的 Actor 对象不会收到任何消息。如果消息发送者本身就是 Actor 对象,那么它就应该使用带两个参数的 `Tell()` 方法,并将它本身的 `Self` 值用作一个参数。非 Actor 对象发送者可以使用带一个参数的 `Tell()` 方法。

小结

我希望 C# 开发者能够通过阅读本附录,更乐于接受 Actor 模型。本书没有列举许多 C# 示例,但你仍旧可以看到在 .NET 平台上使用 Actor 模型的可能性。Dotsero Github 提供了许多测试,展示了 Dotsero 工具集中各种工具的用法。你可以使用 Dotsero 通过你熟悉的 C# 语言重写本书的示例程序。

参考资料

- [2,400-Node Cluster] <http://typesafe.com/blog/running-a-2400-akka-nodes-cluster-on-google-compute-engine>
- [ACM-Amazon] <http://queue.acm.org/detail.cfm?id=1142065>
- [Actor Model] http://en.wikipedia.org/wiki/Actor_model
- [Actor-Endowment] http://en.wikipedia.org/wiki/Object-capability_model
- [Actors-Controversy] http://en.wikipedia.org/wiki/Actor_model#Unbounded_nondeterminism_controversy
- [Actors-Nondeterministic] <http://pchiusano.blogspot.com/2013/09/actors-are-overly-nondeterministic.html>
- [Agha, Gul] Agha, Gul. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: MIT Press, 1986.
- [Akka-Google] <http://typesafe.com/blog/running-a-2400-akka-nodes-cluster-on-google-compute-engine>
- [Akka-Message-Guarantees] <http://doc.akka.io/docs/akka/snapshot/general/message-delivery-guarantees.html>
- [Ambysoft] www.ambysoft.com/surveys/success2013.html
- [AMC-Causal Consistency] <http://queue.acm.org/detail.cfm?id=2610533>
- [Amdahl's law] http://en.wikipedia.org/wiki/Amdahl's_law
- [Atomic-Scala] Eckel, Bruce, and Marsh, Dianne. *Atomic Scala: Learn Programming in a Language of the Future*. Crested Butte, CO: Mindview LLC, 2013.
- [Bolt-on Causal Consistency] www.bailis.org/papers/bolton-sigmod2013.pdf
- [Brewer-Inktomi] <https://www.usenix.org/legacy/publications/library/proceedings/ana97/summaries/brewer.html>
- [CAP] http://en.wikipedia.org/wiki/CAP_theorem
- [Chaos Report] <http://blog.standishgroup.com/post/18>
- [Coursera] www.coursera.org
- [CQRS] <http://martinfowler.com/bliki/CQRS.html>
- [CQS] http://en.wikipedia.org/wiki/Command%E2%80%93query_separation
- [Cray-CDC] http://en.wikipedia.org/wiki/CDC_1604
- [DDD] Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2004.

- [DDJ] www.drdobbs.com/architecture-and-design/2010-it-project-success-rates/226500046
- [Dotsero Toolkit] <https://github.com/VaughnVernon/Dotsero>
- [EDA-Verification] <http://eprints.cs.univie.ac.at/3723/1/4.pdf>
- [EIP] Hohpe, Gregor, and Woolf, Bobby. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston: Addison-Wesley, 2004.
- [False-Sharing] <http://psy-lob-saw.blogspot.com/2014/06/notes-on-false-sharing.html>
- [Fowler EAA] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 2003.
- [GoF] Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995.
- [Google-Architecture] <http://infolab.stanford.edu/~backrub/google.html>
- [Google-Owies] <http://aphyr.com/posts/288-the-network-is-reliable>
- [Google-Platform] http://en.wikipedia.org/wiki/Google_platform
- [Gossip Protocol] http://en.wikipedia.org/wiki/Gossip_protocol
- [Herb Sutter] www.gotw.ca/publications/concurrency-ddj.htm
- [Hewitt-ActorComp] <http://arxiv.org/ftp/arxiv/papers/1008/1008.1459.pdf>
- [Horstmann] www.horstmann.com/scala/index.htm
- [IBM-History] http://en.wikipedia.org/wiki/IBM_7090
- [IBM-zEC12] [http://en.wikipedia.org/wiki/IBM_zEC12_\(microprocessor\)](http://en.wikipedia.org/wiki/IBM_zEC12_(microprocessor))
- [IDDD] Vernon, Vaughn. *Implementing Domain-Driven Design*. Boston: Addison-Wesley, 2013.
- [Inktomi-Architecture] www.thefreelibrary.com/Inktomi+Unveils+Third+Generation+Search+Architecture%3B+Inktomi...-a061423190
- [Intel-FalseSharing] <https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads>
- [Kafka] <http://kafka.apache.org>
- [Kay-DDJ] www.drdobbs.com/architecture-and-design/interview-with-alan-kay/240003442?pgno=3
- [Kay-Squeak] <http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html>

- [Kryo] <https://github.com/EsotericSoftware/kryo>
- [Mackay] www.doc.ic.ac.uk/~nd/surprise_97/journal/vol2/pjm2/
- [Mechanical-Sympathy] <http://mechanical-sympathy.blogspot.com/2011/08/false-sharing-java-7.html>
- [Meyer-OOSC] Meyer, Bertrand. *Object-Oriented Software Construction, Second Edition*. Upper Saddle River, NJ: Prentice Hall, 1997.
- [Moore's Law] http://en.wikipedia.org/wiki/Moore's_law
- [Network-Partition] http://en.wikipedia.org/wiki/Network_partition
- [Nitsan Wakart] <http://psy-lob-saw.blogspot.com/2014/06/notes-on-false-sharing.html>
- [OCM] http://en.wikipedia.org/wiki/Object-capability_model
- [POSA1] Buschmann, Frank et al. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Hoboken, NJ: Wiley, 1996.
- [ProtoBuf] <https://github.com/google/protobuf/>
- [Reactive Manifesto] www.reactivemanifesto.org/
- [Read-Write] readwrite.com/2014/07/10/akka-jonas-boner-concurrency-distributed-computing-internet-of-things
- [Retlang] <https://code.google.com/p/retlang/>
- [Retlang-CTX] http://www.jroller.com/mrettig/entry/lightweight_concurrency_in_net_similar
- [Roestenburg] <http://doc.akka.io/docs/akka/snapshot/scala/testkit-example.html>
- [sbt-Suereth] Suereth, Josh and Farwell, Matthew. *SBT in Action: The simple Scala build tool*. Shelter Island, NY: Manning Publications, 2015.
- [ScalaTest] www.scalatest.org/
- [SIS] http://en.wikipedia.org/wiki/Strategic_information_system
- [Split-Brain] [https://en.wikipedia.org/wiki/Split-brain_\(computing\)](https://en.wikipedia.org/wiki/Split-brain_(computing))
- [SRP] http://en.wikipedia.org/wiki/Single_responsibility_principle
- [Suereth] Suereth, Joshua. *Scala in Depth*. Shelter Island, NY: Manning Publications, 2012.
- [Tilkov-CDM] <https://www.innoq.com/en/blog/thoughts-on-a-canonical-data-model/>
- [Transistor Count] http://en.m.wikipedia.org/wiki/Transistor_count
- [Transistor] http://en.wikipedia.org/wiki/History_of_the_transistor

[Triode] <http://en.wikipedia.org/wiki/Triode>

[Typesafe] <http://typesafe.com>

[Vogels-AsyncArch] www.webperformancematters.com/journal/2007/8/21/asynchronous-architectures-4.html

[Vogels-Scalability] www.allthingsdistributed.com/2006/03/a_word_on_scalability.html

[Westheide] www.parleys.com/play/53a7d2c6e4b0543940d9e54d

[WhitePages] http://downloads.typesafe.com/website/casestudies/Whitepages-Final.pdf?_ga=1.150961718.1894161222.1397685553

企业级软件开发工作变得越来越困难并且极易失败。资深的软件开发工程师和计算机畅销图书作者Vaughn Vernon, 为大家展示了怎样通过Actor模型编写更简单和回报率更高的成功软件。本书介绍了编写响应式企业级软件的方式, 还介绍了如何使用Actor模型、Scala语言和Akka框架等功能强大的工具, 突破性能极限、获得高可伸缩性和巧妙地处理最具挑战性的非功能性需求的方式。

通过翔实反映作者在尖端编程领域所做的工作, Vaughn Vernon向架构师和开发者们展示了将长时间以来得到业内人士肯定的Actor模型理论运用到实践中的方式。首先, 本书介绍了响应式软件的宗旨, 以及通过由消息驱动的Actor模型, 使系统拥有更高的响应性、韧性和弹性的方式。然后, 本书介绍了Scala语言的基础知识, 概要介绍了Akka框架和Akka集群功能, 还使用一整章的篇幅介绍了通过Scala语言和Akka框架最大化程序的性能和可伸缩性的方式。

掌握了这些基础知识后, 读者就可以学习通过创建消息通道和端点编写和整合企业级软件的知识。之后, 本书还介绍了构建和转换消息, 以及为消息提供路由的方式, 使读者能够轻松地编写出更为成功的程序。

Vaughn Vernon是一位资深的软件开发者, 并且是一位简化软件设计和实现思想的领袖人物。他是畅销书《实现领域驱动设计》(已由电子工业出版社在2014年出版)的作者。他还为来自世界各地的数百位软件开发者教授IDDD Workshop课程。Vaughn Vernon经常在计算机行业大会上发表演讲。他擅长的领域包括分布式计算、消息传输模式和Actor模型。2012年, 他在一个GIS系统中第一次使用了Akka框架。此后, 他就一直专门研究通过由领域驱动的设计模式应用Actor模型的技术。通过关注Vaughn Vernon的博客(www.VaughnVernon.co)和微博(Twitter网站的@VaughnVernon用户), 可以了解他的最新著作。

本书包括以下内容:

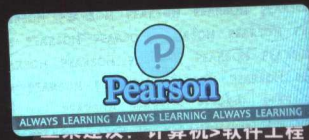
- ◎使用响应式架构通过全方位简化编程工作的各个层面, 消除编程复杂性的方式
- ◎Actor系统和Actor对象的特点, 以及通过Akka框架使它们拥有更强大功能的方式
- ◎怎样创建可以在一个或多个计算节点中自由伸缩的系统
- ◎创建通道机制的方式, 以及怎样选择适当的通道解决应用程序的开发和整合难题
- ◎应使消息拥有怎样的结构, 才能将消息发送者的意图清晰地告知消息接收者
- ◎为由领域驱动的应用程序实现处理过程管理器的方式
- ◎降低消息源和消息目的地之间的耦合性, 和将适当的业务逻辑整合到消息路由器中的方式
- ◎在应用程序开发和整合环境中会用到的各种消息转换方式
- ◎使用事件溯源模式和符合CQRS原则的响应式视图, 实现持久化Actor对象的方式



策划编辑: 张春雨
责任编辑: 刘 舫
封面设计: 李 玲



Pearson



ISBN 978-7-121-29113-5



定价: 99.00元